algorithm-note

{ Programming Ability Test

- 推荐使用 **code::blocks**, **C-Free**, **dev-c++**
- 目录结构 `评测系统名/题号`

算法归纳

排序

- 选择排序

```
void selectSort(){
 for(int i = 0;i < n;i++){
     int k = i;
     for(int j = i;i < n;j++){
         if(a[j] < a[k]){
             k = j;
         }
     }
     swap(a[i],a[k]);
 }
}
```

- 插入排序

```
void insertSort(){
    for(int i = 1;i < n;i++){
        int temp = a[i], j = i;
        while(j > 0 && temp < a[j-1]){
            a[j] = a[j-1];
            j--;
        }
        a[j] = temp;
    }
}
```

- 排序题与sort函数的应用

散列

- 整数散列
- 字符串hash（ASCII码表示）

递归

- 全排列

```
#include <cstdio>
const int maxn = 11;
int n,P[maxn],hashTable[maxn] = {false};

void generateP(int index){
    if(index == n+1){
        for(int i = 1;i <= n;i++){
            printf("%d", P[i]);
        }
        printf("\n");
        return;
    }
    for(int x = 1;x <= n;x++){
        if(hashTable[x] == false){
            P[index] = x;
            hashTable[x] = true;
            generateP(index+1);
            hashTable[x] = false;
        }
    }
}

int main(){
    n = 3; // 1~3的全排列
    generateP(1);
    return 0;
}
```

- n皇后问题

```
void generateP(int index){
    if(index == n+1){
        count++;
        return;
    }
    for(int i = 1;i <= n;i++){
        if(hashTable[i] == false){
            bool flag = true;
            for(int pre = 1;pre < index;pre++){
                if(abs(pre-index) == abs(i-p[pre])){
                    flag = false;
                    break;
                }
            }
            if(flag){
                p[index] = i;
                hashTable[i] = true;
                generateP(index+1);
                hashTable[i] = false;
            }
        }
    }
}
```

贪心

- 简单贪心
- 区间贪心

```
struct Inteval{
    int x,y;
}I[maxn];
bool cmp(Inteval a, Inteval b){
    if(a.x != b.x) return a.x > b.x;
    else return a.y < b.y;
}
// 关键代码
sort(I, I+n, cmp);
int ans = 1, lastX = I[0].x;
for(int i = 1;i < n;i++){
    if(I[i].y <= lastX){
        lastX = I[i].x;
        ans++; // ans保留不相交区间条数
    }
}
```

二分

```
// 基本框架
int mid;
while(left < right){
    mid = (left + right) / 2;
    // 与a[mid]作比较
    // 自定义处理逻辑
}
```

- 二分查找
- 二分法拓展
    - `f(x)=x²` 计算根号2的值
    - 半圆形储水装置的装水问题
    - 木棒切割问题（段数一定的情况下每段最长多长）
- 快速/二分幂

```
// 计算a^b % m
typedef long long LL;

// 递归写法
LL binaryPow(LL a, LL b, LL m){
  if(b == 0) return 1;
  if(b % 2 == 1) return a*binaryPow(a, b-1, m) % m;
  else{
    LL mul = binaryPow(a, b / 2, m);
    return mul * mul % m;
  }
}

// 迭代写法
LL binaryPow(LL a, LL b, LL m){
  LL ans = 1;
  while(b > 0){
    if(b & 1){
      ans = ans * a % m;
    }
    a = a * a % m;
    b >>= 1;
  }
}
```

two pointers

体会一下下面这段代码，寻找`a[i]+a[j]=M`的组合。

```
// a[x]有序
while(i < j){
    if(a[i] + a[j] == M){
        cout << i,j << endl;
        i++;
        j--;
    }else if(a[i] + a[j] < M){
        i++;
    }else{
        j--;
    }
}
```

- 2-路归并排序

```
// recursive
void mergeSort(int a[], int left, int right){
    if(left < right){
        int mid = (left + right) / 2;
        mergeSort(a, left, mid);
        mergeSort(a, mid+1, right);
        merge(a, left, mid, mid+1, right);// 合并[left,mid],[mid+1,right]
    }
}

// 非递归*
void mergeSort(int a[]){
    for(int step = 2;step /2 <= n;step *= 2){
        for(int i = 1;i <= n;i += step){
            int mid = i + step / 2 - 1;
            if(mid + 1 <= n){
                merge(a, i, mid, mid+1, min(i+step-1,n));
            }
        }
    }
}

void merge(int a[], int L1, int R1, int L2, int R2){
    int i = L1, j = L2;
    int temp[maxn], index = 0;
    while(i <= R1 && j <= R2){
        if(a[i] <= a[j]){
            temp[index++] = a[i++];
        }else{
            temp[index++] = a[j++];
        }
    }
    while(i <= R1) temp[index++] = a[i++];
    while(j <= R2) temp[index++] = a[j++];
    for(int i = 0;i < index;i++){
        a[L1+i] = temp[i];
    }
}
```

- 快速排序

```c
void quickSort(int a[], int left, int right){
    if(left < right){
        int pos = partition(a, left, right);
        quickSort(a, left, pos);
        quickSort(a, pos+1, right);
    }
}

int partition(int a[], int left, int right){
    int temp = a[left];
    while(left < right){
        while(left < right && a[right] > temp) right--;
        a[left] = a[right];
        while(left < right && a[left] <= temp) left++;
        a[right] = a[left];
    }
    a[left] = temp;
    return left;
}

// =>randPartition
#include <stdlib.h>
#include <time.h>
int main(){
    srand((unsigned)time(NULL));
    // ...
}
int randPartition(int a[], int left, int right){
    int p = (round(rand()*1.0 / RAND_MAX * (right - left)) + left);
    swap(a[p],a[left]);
    int temp = a[left];
    while(left < right){
        while(left < right && temp < a[right]) right--;
        a[left] = a[right];
        while(left < right && temp >= a[left]) left++;
        a[right] = a[left];
    }
    a[left] = temp;
    return left;
}
```

数学问题

数字黑洞

主要是`to_array`、`to_number`函数的编写

```
void to_array(int n, int num[]){
    for(int i = 0;i < 4;i++){
        num[i] = n % 10;
        n /= 10;
    }
}

int to_number(int num[]){
    int sum = 0;
    for(int i = 0;i < 4;i++){
        sum = num[i] + sum * 10;
    }
    return sum;
}
```

最大公约数和最小公倍数

```
// 最大公约数gcd
int gcd(int a, int b){
    if(b == 0) return a;
    else return gcd(b, a%b);
}

// 最小公倍数
int lcm(int a, int b){
    return a / gcd(a,b) * b; // 防止a*b溢出，所以写成a/gcd*b
}
```

素数

```
// 1不是素数
bool isPrime(int a){
  if(a <= 1) return false;
  int sqr = (int)sqrt(1.0*a);
  for(int i = 2;i <= sqr;i++){
      if(a % i == 0) return false;
  }
  return true;
}
```

- 埃氏筛法-寻找素数表

```
// 时间复杂度O(nloglogn)
const int maxn = 101; // 表长
int prime[maxn], pNum = 0;
bool p[maxn] = {0};
void Find_Prime(){
    for(int i = 2;i < maxn;i++){
        if(p[i] == false){
            prime[pNum++] = i;
            for(int j = i + i;j < maxn;j += i){
                p[j] = true;
            }
        }
    }
}
```

质因子分解

```
struct factor{
    int x, cnt; // x为质因子，cnt为其个数
}fac[10];

if(n % prime[i] == 0){
    fac[num].x = prime[i];
    fac[num].cnt = 0;
    while(n % prime[i] == 0){
        fac[num].cnt++;
        n /= prime[i];
    }
    num++;
}

if(n != 1){
    fac[num].x = n;
    fac[num++].cnt = 1;
}
```

大整数运算

```
// big number结构体表示
struct bign{
    int d[1000];
    int len;
    bign(){ // 结构体的构造函数
        memset(d,0,sizeof(d)); // 相当于高位自动填充0
        len = 0;
    }
}
```

- 高精度（大整数）加减法

```
bign add(bign a, bign b){
    bign c;
    int carry = 0;
    for(int i = 0;i < a.len || i < b.len;i++){
        int temp = a.d[i] + b.d[i] + carry;
        c.d[c.len++] = temp % 10;
        carry = temp / 10;
    }
    if(carry != 0){
        c.d[c.len++] = carry;
    }
    return c;
}
bign sub(bign a, bign b){ // a>=b
    bign c;
    for(int i = 0;i < a.len || i < b.len;i++){
        if(a.d[i] < b.d[i]){
            a.d[i+1]--;
            a.d[i] += 10;
        }
        c.d[c.len++] = a.d[i] - b.d[i];
    }
    while(c.len > 1 && c.d[c.len - 1] == 0){
        c.len--; // 至少要有一位
    }
    return c;
}
```

组合数

- n!中有多少个质因子p

```
// (n/p + n / p^2 + n / p^3 + ···)
int cal(int n, int p){
    int ans = 0;
    while(n){
        ans += n / p;
        n /= p;
    }
    return ans;
}
// 若要求n!末尾有几个零，可以转换为有多少个质因子5的问题
// 递归版本
int cal(int n, int p){
    if(n < p) return 0;
    return n / p + cal(n / p, p);
}
```

- 组合数的计算

直接按定义算容易超出数据范围，即使是`long long`类型也只能接受`n<=20`的运算，不做阐述。所以利用下面这个公式可以写出递归函数。

$$C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$$

```
// 递归
long long res[67][67] = {0};
long long C(long long n, long long m){
    if(m == 0 || m == n) return 1;
    if(res[n][m] != 0) return res[n][m]; // 避免重复计算
    return res[n][m] = C(n-1,m) + C(n-1,m-1);
}

// 递推
const int n = 60;
void calC(){
    for(int i = 1;i <= n;i++){
        res[i][0] = res[i][i] = 1;
    }
    for(int i = 2;i <= n;i++){
        for(int j = 0;j <= i/2;j++){
            res[i][j] = res[i-1][j] + res[i-1][j-1];
            res[i][i-j] = res[i][j];
        }
    }
}
```

或者另法

$$C_n^m = \frac{n!}{m!(n-m)!} = \frac{(n-m+1)(n-m+2)...(n-m+n)}{1*2*3*...*m}$$

```
// O(m)时间复杂度，excellent
long long C(long long n, long long m){
    long long ans = 1;
    for(long long i = 1;i <= m; i++){
        ans = ans * (n-m+i) / i; // 一定要先乘再除，保证整除
    }
    return ans;
}
```

C++标准模板库STL

注意：涉及到的区间问题通通都是左闭右开

使用STL库需要加一句`using namespace std;`

- vector `#include <vector>`

```
vector<typename> a; //typename也可以是stl容器，比如
vector< vector<int> > name; //可以理解为二维数组，注意空格
vector<int>::iterator it; // 迭代器，可以理解为指针，使用*it可以访问元素
for(vector<int>::iterator it = vi.begin();it != vi.end();it++){
    // ...
}
// 注意：只有vector和string可以使用vi.begin()+3这样的写法
push_back()
pop_back()
size()
clear()
insert(it, x)
erase(it) // 删除单个
erase(first, last) // 删除区间，左闭右开
```

- set `#include <set>`

```
set<typename> a; // 元素自动递增排序，去除重复元素
insert(x)
set<int>::iterator it = st.find(x) // 返回迭代器 print *it
erase(it)
erase(value)
erase(first, last)
size()
clear()
```

- string `#include <string>`

```
string str; // 可以用下标访问，也可以用迭代器访问
cin >> str; cout << str; printf("%s",str.c_str()); // 输入输出
string::iterator it;
str3 = str1 + str2; //拼接+=，还可以比大小，比较规则是字典序
length()/size() // 都可以，基本相同
insert(pos, string) // 在str[pos]处插入string
insert(it, it2, it3) // 与插入位置it，待插字符串的首位迭代器[it2,it3)，同样左闭右开
erase(it)
erase(first, last)
erase(pos, length)
clear()
substr(pos, len)
string::npos // find()失败的返回值
find(str2) // 若存在，返回str2在str中第一次出现的位置
find(str2, pos) // 指定位置
replace(pos, len, str2)
replace(pos, it1, it2)
```

- map `#include <map>`

```
map<typename1, typename2> mp;// 以key的大小递增排序
map<typename1, typename2>::iterator it;// it->first it->second
find(key)
erase(it)
erase(key)
erase(first, last)
size()
clear()
```

- queue `#include <queue>`

```
queue<typename> name;
name.front(); name.back();
push()
pop()
empty()
size()
```

- priority_queue `#include <queue>`

```
priority_queue<typename> name; // 队首元素是优先级最大的
push()
top()
pop()
empty()
size()
// 基本数据类型优先级设置
priority_queue<int> q; // 等价于
priority_queue<int, vector<int>, less<int> > q; // less表示数字大的优先级越大，greater相反
// 结构体优先级设置
struct fruit{
    string name;
    int price;
    friend bool operator < (fruit f1, fruit f2){ // 重载 <
        return f1.price < f2.price; // 价格高的优先级高，与sort正好相反
        // return f1.price > f2.price; // 价格低的优先级高
    }
}
```

- stack `#include <stack>`

```
stack<typename> name;
push()
top()
pop()
empty()
size()
```

- pair `#include <utility>或者#include <map>`

```
pair<typename1,typename2> name; //可以看作是内部有两个元素的结构体
pair<string, int> p("hh",5); // 初始化
// 临时构建pair赋值,可以用于插入map的键值对
pair<string, int>("hh",5);
make_pair("hh",5);
// 访问元素
p.first p.second //比较大小的话先比first再比second
```

- algorithm头文件下的常用函数
    - `max,min,abs(abs参数必须是整数，浮点数用math头文件下的fabs)`
    - `swap(x,y)`
    - `reverse`

- `next_permutation(给出一个序列在全排列中的下一个序列)`
- `fill`
- `sort`
- `lower_bound,upper_bound(用在有序数组或容器中)`

搜索

深度优先搜索DFS

```
#include <cstdio>
const int maxn = 30;
int n,V,maxValue = 0; // 物品件数n，背包容量V，最大价值maxValue
int w[maxn],c[maxn]; // 重量，价值
// 时间复杂度O(2^n)，不好
void DFS(int index, int sumW, int sumC){
    if(index == n){
        if(sumW <= V && sumC > maxValue){
            maxValue = sumC;
        }
        return;
    }
    DFS(index+1,sumW,sumC);// 不选第index件物品
    DFS(index+1,sumW+w[index],sumC+c[index]);
}
// 优化，剪枝
void DFS(int index, int sumW, int sumC){
    if(index == n) return;
    DFS(index+1,sumW,sumC);
    if(sumW + w[index] <= V){
        if(sumC + c[index] > maxValue){
            maxValue = sumC + c[index];
        }
        DFS(index+1,sumW+w[index],sumC+c[index]);
    }
}
```

广度优先搜索BFS

```
void BFS(int s){
    queue<int> q;
    q.push(s);
    while(!q.empty()){
        // 取出队首元素top
        // 访问队首元素top
        // 将队首元素出队
        // 将top的下一层节点中未曾入队的节点全部入队，并设置为已入队
    }
}
```

最短路径

单点最短路径

Dijkstra算法

```
// G为图，d为源点到各点的距离，s为起点
Dijkstra(G[], d[], s){
    初始化;
    for(循环n次){
        u = 使d[u]最小的还未被访问的顶点的标号;
        记u已被访问;
        for(从u出发能到达的所有顶点v){
            if(v未被访问&&以u为中介点使s到v的最短距离d[v]更优){
                优化d[v];
            }
        }
    }
}
```

Bellman-Ford算法（BF算法）和SPFA算法

> Dijkstra算法不能处理负权图而BF算法可以

O(n^3)的复杂度限制了顶点数约在200以内，因此用邻接矩阵来实现比较合适。

```
// Bellman-Ford O(VE)
for(int i = 0;i < n - 1;i++){ // n为顶点数
    for(each edge u->v){ // 每轮都遍历所有边
        if(d[u] + length[u->v] < d[v]){
            d[v] = d[u] + length[u->v]; // 松弛操作
        }
    }
}
// 判断负环
for(each edge u->v){ // 对所有边在进行一轮遍历
    if(d[u] + length[u->v] < d[v]){ // 如果仍可以被松弛
        return false; // 说明图中有从源点可达的负环
    }
    return true; // 数组d所有值都已达到最优
}

// SPFA 优化后的Bellman
queue<int> Q;
源点s入队;
while(队列非空){
    取出队首元素;
    for(u的所有邻接边u->v){
        if(d[u] + dis < d[v]){
            d[v] = d[u] + dis;
            if(v当前不在队列){
                v入队;
                if(v入队次数大于n-1){
                    说明有可达负环，return;
                }
            }
        }
    }
}
```

全源最短路径

Floyd算法

O(n^3)的复杂度限制了顶点数约在200以内，因此用邻接矩阵来实现比较合适。

```
枚举顶点k ∈ [1, n]
  以顶点k作为中介点，枚举所有顶点对i和j(i ∈ [1, n], j ∈ [1, n])
    如果dis[i][k] + dis[k][j] < dis[i][j] 成立
      赋值dis[i][j] = dis[i][k] + dis[k][j]
```

最小生成树

prim算法（适用于稠密图，边多）

```
// G为图，一般设成全局变量；数组d为顶点与集合S的最短距离
Prim(G, d[]){
    初始化; // 这里可以默认0号节点作为起始点,d[0] = 0
    for(循环n次){
        u = 使d[u]最小的还未被访问的顶点的标号;
        记u已被访问;
        for(从u出发能到达的所有顶点v){
            if(v未被访问&&以u为中介点使得v与集合S的最短距离d[v]更优){
                将G[u][v]赋值给v与集合S的最短距离d[v]; // 与dijkstra算法就差在这里，其他都一样
            }
        }
    }
}
```

kruskal算法（适用于稀疏图，边少）

并查集 Union Find Set

```
int kruskal(){
    令最小生成树的边权之和为ans、最小生成树的当前边数Num_Edge;
    将所有边按边权从小到大排序;
    for(从小到大枚举所有边){
        if(当前测试边的两个端点在不同的连通块中){
            将该测试边加入最小生成树中;
            ans += 测试边的边权;
            最小生成树的当前边数Num_Edge加1;
            当边数Num_Edge等于顶点数减1时结束循环;
        }
    }
}
```

动态规划DP

状态转移方程，以经典的数塔问题为例

```
// 边界
for(int j = 1;j <= n;j++){
    dp[n][j] = f[n][j];
}
// 自底向上
for(int i = n - 1;i >= 1;i--){
    for(int j = 1;j <= i;j++){
        dp[i][j] = max(dp[i+1][j], dp[i+1][j+1]) + f[i][j];
    }
}
// print dp[1][1]
```

最大连续子序列和（只考虑结尾元素）

状态转移方程：`dp[i] = max{A[i], dp[i-1] + A[i]}`

边界：`dp[0] = A[0]`

最长不下降子序列LIS（同样只考虑以A[i]结尾的情况）

状态转移方程：`dp[i] = max{1, dp[j] + 1} (j = 1,2,...,i-1 && A[j] <= A[i])`

```
int ans = -1; // 记录最大dp
for(int i = 1;i <= n;i++){
    dp[i] = 1; // 初始假设每个元素自成序列
    for(int j = 1;j <i; j++){
        if(A[i] >= A[j] && (dp[j] + 1 > dp[i])){
            dp[i] = dp[j] + 1;
        }
    }
    ans = max(ans, dp[i]);
}
```

最长公共子序列LCS

状态转移方程：`当A[i] == B[i], dp[i][j] = dp[i-1][j-1] + 1`

`当A[i] != B[i], dp[i][j] = max{dp[i-1][j], dp[i][j-1]}`

边界：`dp[i][0] = dp[0][j] = 0`

```
// 边界
for(int i = 0;i <= lenA;i++) dp[i][0] = 0;
for(int j = 0;j <= lenB;j++) dp[0][j] = 0;
// 状态转移方程
for(int i = 1;i <= lenA;i++){ // 下标从1开始
    for(int j = 1;j <= lenB;j++){
        if(A[i] == B[i]){
            dp[i][j] = dp[i-1][j-1] + 1;
        }else{
            dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
}
```

最长回文子串

状态转移方程： `dp[i][j] = dp[i+1][j-1], S[i] == S[j]`

`dp[i][j] = 0, S[i] != S[j]`

边界： `dp[i][i] = 1, dp[i][i+1] = (S[i] == S[i+1])?1:0`

```
// 边界
ans = 1;
memset(dp, 0, sizeof(dp));
for(int i = 0;i < len;i++){
    dp[i][i] = 1;
    if(i < len - 1){
        if(S[i] == S[i+1]){
            dp[i][i+1] = 1;
            ans = 2;
        }
    }
}
// 状态转移方程,先计算长度为3的dp值，然后是长度为4,5,....
for(int L = 3;L <= len;L++){
    for(int i = 0;i + L - 1 < len;i++){
        int j = i + L - 1;
        if(S[i] == S[j] && dp[i+1][j-1] == 1){
            dp[i][j] = 1;
            ans = L;
        }
    }
}
```

DAG最长路

```
// 从i号点出发能得到的最长路径,初始化dp数组为0
int DP(int i){
    if(dp[i] > 0) return dp[i]; // dp[i]已计算得到
    for(int j = 0;j < n;j++){
        if(G[i][j] != INF){
            dp[i] = max(dp[i], DP(j) + G[i][j]); // 出度为0的点dp[i] = 0
        }
    }
    return dp[i];
}

// 从i号顶点出发到达终点T能获得的最长路径长度,初始化dp数组为-INF
int DP(int i){
    if(vis[i]) return dp[i];
    vis[i] = true;
    for(int j = 0;j < n;j++){
        if(G[i][j] != INF){
            dp[i] = max(dp[i], DP(j) + G[i][j]); // 边界dp[T] = 0
        }
    }
    return dp[i];
}
```

背包问题

01背包问题

状态转移方程： `dp[i][v] = max{dp[i-1][v], dp[i-1][v-w[i]]+c[i]} (1 <= i <= n,w[i] <= v <= V)`

边界： `dp[0][v] = 0, 0 <= v <= V`

```
for(int i = 1;i <= n;i++){
    for(int v = w[i];v <= V;v++){
        dp[i][v] = max(dp[i-1][v],dp[i-1][v-w[i]] + c[i]);
    }
}

// 空间复杂度优化->滚动数组，一维
for(int i = 1;i <= n;i++){
    for(int v = V;v >= w[i];v--){ // 注意是逆序!
        dp[v] = max(dp[v], dp[v-w[i]]+c[i]);
    }
}
```

完全背包问题

状态转移方程：`dp[i][v] = max(dp[i-1][v], dp[i][v-w[i]]+c[i])`

边界：`dp[0][v] = 0`

```
// 一维形式的状态转移方程和10背包问题一模一样，只是用正序遍历V,区分清楚为什么
for(int i = 1;i <= n;i++){
    for(int v = w[i];v <= V;v++){
        dp[v] = max(dp[v], dp[v-w[i]]+c[i]);
    }
}
```

tips

- 计算日期差值可以定义一个二维数组存放月份天数`int month[13][2]`，下标作为月份，每一维存放平年和闰年的天数，`{{0,0},{31,31},{28,29},...}`，闰年`(year % 4 == 0 && year % 100 != 0 || year % 400 == 0)`
- 回文串判定`str[i] == str[len - i - 1]`
- 坐标变化可以定义两个数组`x[] = {0， 0， -1， 1}`和`y[] = {1， -1， 0， 0}`
- 浮点数比较是否相等不能用`==`，使用`fabs(a-b) < eps`，eps根据题目取对应精度，一般取`1e-8`