

# 《全国青少年信息学奥林匹克系列竞赛大纲》 知识点整理

---

## 《全国青少年信息学奥林匹克系列竞赛大纲》知识点整理

### 2.1 入门级

#### 2.1.1 基础知识与编程环境

#### 2.1.2 C++ 程序设计

##### 1、程序基本概念

编辑、编译、解释、调试的概念

编辑

编译

解释

调试

##### 2、基本数据类型

##### 3、程序基本语句

##### 4、基本运算

##### 5、数学库常用函数

绝对值函数、四舍五入函数、下取整函数、上取整函数、平方根函数、常用三角函数、对数函数、

指数函数

绝对值函数 `abs(x)`

四舍五入函数 `round(x)`

下取整函数 `floor(x)`

上取整函数 `ceil(x)`

平方根函数 `sqrt(x)`

常用三角函数

对数函数

指数函数

##### 6、结构化程序设计

##### 7、数组

##### 8、字符串的处理

##### 9、函数与递归

##### 10、结构体与联合体

##### 11、指针类型

##### 12、文件及基本读写

##### 13、STL 模板

栈 (stack)、队列 (queue)、链表 (list)、向量 (vector) 等容器

#### 2.1.3 数据结构

#### 2.1.4 算法

##### 1、算法概念与描述

算法描述：自然语言描述、流程图描述、伪代码描述

自然语言描述

流程图描述

伪代码描述

##### 2、入门算法

##### 3、基础算法

倍增法 (binary lifting)

ST 算法

最近公共祖先 (LCA)

##### 4、数值处理算法

##### 5、排序算法

- 计数排序
  - 概念
  - 总结
  - 代码
  - 应用场景
- 6、搜索算法
- 7、图论算法
  - 泛洪算法 (flood fill)
    - 作用
    - 基本思想
    - 算法参数
    - 算法实现
- 8、动态规划
  - 简单区间类型动态规划
    - 例子
    - 代码
    - 代码演示过程
- 2.1.5 数学与其他
  - 1、数及其运算
  - 2、初等数学
  - 3、初等数论
    - 素数筛法：埃氏筛法与线性筛法
      - 暴力筛法
      - 埃式筛
      - 欧拉筛
  - 4、离散与组合数学
  - 5、其他
    - 格雷码
      - 认识格雷码
      - 位元
      - 码值
      - 格雷码的生成
      - 二进制转格雷码
- 2.2 提高级
  - 2.2.1 基础知识与编程环境
  - 2.2.2 C++ 程序设计
    - 1、类 (class)
      - 类的概念及简单应用
        - 类的概念
        - 简单应用
      - 成员函数和运算符重载
    - 2、STL 模板
      - 容器 (container) 和迭代器 (iterator)
      - 对 (pair) 、元组 (tuple)
      - 集合 (set) 、多重集合 (multiset)
      - 双端队列 (deque) 、优先队列 (priority\_queue)
      - 映射 (map) 、多重映射 (multimap)
      - 算法模板库中的常用函数
  - 2.2.3 数据结构
    - 1、线性结构
    - 2、集合与森林
      - 树的孩子兄弟表示法
        - 左孩子右兄弟表示的树的高度
        - 左孩子右兄弟表示的树的叶子结点数
        - 树的总结点数

- 树的先根遍历
- 树的后根遍历
- 左孩子右兄弟表示的树的层次遍历
- 左孩子右兄弟表示的树的宽度
- 在以t为根的树中找结点p的双亲

### 3、特殊树

- 树状数组
  - 概念
  - 前置知识
  - 问题引入
  - 树状数组结构分析
  - 单点修改，区间查询
  - 区间修改，单点查询
- 笛卡尔树
  - 定义
  - 性质
  - 例子
  - 代码
- 平衡树：AVL、treap、splay 等
  - 平衡二叉树（AVL）
  - Treap树（树堆）
  - 伸展树（Splay）

### 4、常见图

- 偶图（二分图）(bipartite graph)
  - 偶图的定义
  - 完全偶图（完全二分图）的定义
  - 偶图的充分必要条件
  - 匹配的引入
  - 偶图的匹配
  - 匹配的判定条件（霍尔定理，t条件）
  - 匹配的应用
- 欧拉图
  - 基本概念
  - 起源历史
  - 相关定理
- 双连通图
  - 概念

### 5、哈希表

- 哈希法概念
- 哈希法思想
- 哈希冲突
- 优缺点
- 构造原则
- 数值哈希函数构造、哈希冲突的常用处理方法
  - 哈希函数的构造方法
  - 解决冲突的方法
  - 结论
- 字符串哈希函数构造

#### 2.2.4 算法

- 1、复杂度分析
  - 时间复杂度分析
    - 大O复杂度表示法
    - 常见时间复杂度与增长速度
    - 最好、最坏与平均时间复杂度
- 2、算法策略

- 3、基础算法
- 4、排序算法
  - 基数排序
    - 算法思想
    - 分类
    - 图示
    - 代码实现
    - 运行结果
    - 时间复杂度&空间复杂度&稳定性
- 5、字符串相关算法
  - 字符串匹配：KMP 算法
    - 背景
    - 真前缀和真后缀
    - KMP字符串匹配算法
- 6、搜索算法
  - 搜索的剪枝优化
    - 1. 优化搜索顺序
    - 2. 排除等效冗余
    - 3. 可行性剪枝
    - 4. 最优化剪枝
  - 实践应用
  - 记忆化搜索
    - 概念
    - 记忆化搜索与递推区别
    - 记忆化搜索解题步骤
    - 相关习题
  - 启发式搜索
    - 概念
    - A\*搜索
    - 预备知识
    - 正题
    - 思考
    - 小结
  - 双向广度优先搜索
    - 概念
    - 伪代码
  - 迭代加深搜索
    - 概念
    - 伪代码
    - 迭代加深搜索的优点和局限性
- 7、图论算法
  - 次小生成树
    - 回顾最小生成树算法——Kruskal 算法
    - 非严格次小生成树
    - 严格次小生成树
  - 单源次短路
    - 单源最短路径
    - 非严格次短路
    - 严格次短路
  - 二分图的判定
    - 染色法+DFS
    - 并查集
    - 例子
  - 割点、割边
  - 树的重心、直径、DFS 序与欧拉序

树上差分、子树和与倍增  
8、动态规划  
    状态压缩动态规划  
    动态规划的常用优化  
2.2.5 数学与其他  
    1、初等数学  
        代数（高中部分）  
        几何（高中部分）  
    2、初等数论  
        同余式  
        欧拉定理和欧拉函数  
        费马小定理  
        威尔逊定理  
        裴蜀定理  
        模运算意义下的逆元  
        扩展欧几里得算法  
        中国剩余定理  
    3、离散与组合数学  
        多重集合  
        等价类  
        多重集上的排列  
        多重集上的组合  
        错排列、圆排列  
        鸽巢原理  
        二项式定理  
        容斥原理  
        卡特兰（Catalan）数  
    4、线性代数  
        向量与矩阵的概念  
        向量的运算  
        矩阵的初等变换  
        矩阵的运算：加法、减法、乘法与转置  
        特殊矩阵的概念：单位阵、三角阵、对称阵和稀疏矩阵  
        高斯消元法

### 2.3 NOI 级

#### 课程进度：

[2.1.2 \(1\)](#) — [2.1.4 \(7\)](#) 共7部分

[2.1.4 \(8\)](#) — [2.2.3 \(3\)](#) 共9部分

[2.2.3 \(4\)](#) — [2.2.4 \(1\)](#) 共10部分

[2.2.4 \(4\)](#) — [2.2.4 \(7\)](#) 共10部分

## 2.1 入门级

### 2.1.1 基础知识与编程环境

## 2.1.2 C++ 程序设计

### 1、程序基本概念

#### 编辑、编译、解释、调试的概念

##### 编辑

编辑是指编写程序代码的过程。程序员使用文本编辑器或集成开发环境（IDE）来创建、编辑和保存程序源代码文件。在编辑代码时，程序员可以使用代码补全、语法高亮、代码折叠等功能来提高效率。

##### IDE:

集成开发环境，一般包括代码编辑器、编译器、调试器和图形用户界面工具。集成了代码编写功能、分析功能、编译功能、调试功能等一体化的开发软件服务套。所有具备这一特性的软件或者软件套(组)都可以叫集成开发环境。

每种编程语言都有一些特定的IDE，如学习C/C++/C# 推荐微软强大的Visual Studio系列，CodeBlocks等；学习JAVA推荐Eclipse, IntelliJ Idea和NetBeans等；学习Python 推荐 PyCharm, WingIDE, python官方支持的集成开发环境idle（轻量级）等。

##### 编译

编译是将程序源代码转换为可执行文件的过程。编译器是用于执行编译过程的工具，它将源代码转换为机器代码，这些机器代码可以在计算机上执行。编译器通常会进行词法分析、语法分析、语义分析等操作来检查代码的正确性，并生成可执行代码。

##### 编译器：

把源代码转换成（翻译）低级语言的程序，一个现代编译器的主要工作流程：源代码（source code）→ 预处理器（preprocessor）→ 编译器（compiler）→ 目标代码（object code）→ 链接器（Linker）→ 可执行程序（executables）。

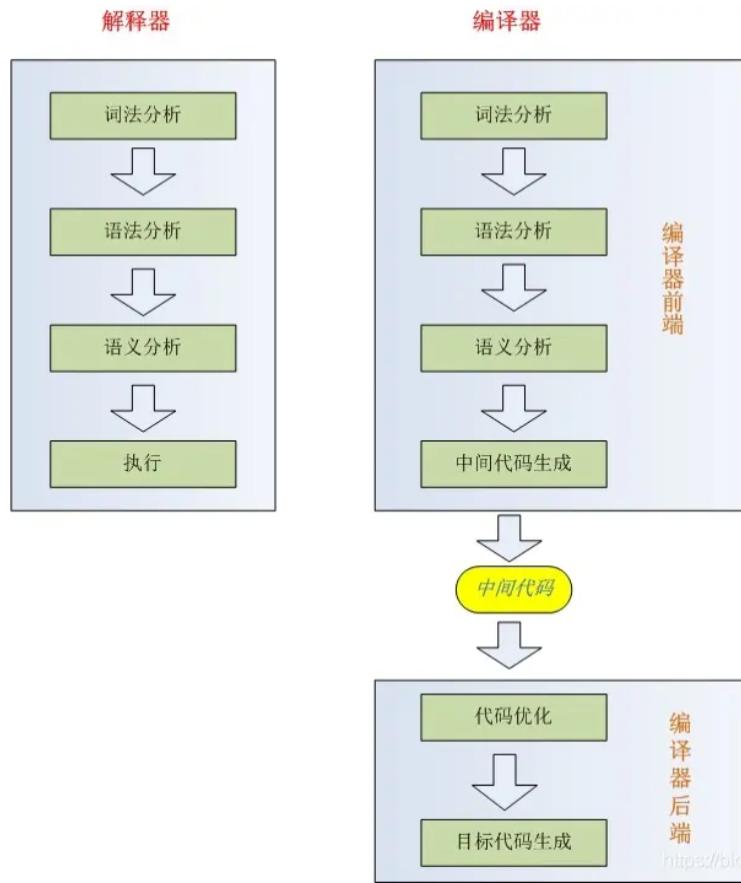
常用编译器如Dev-C++, WIN-TC, TurboC, MinGW等。

源代码一般为高级语言（High-level language），如Pascal、C、C++、Java、汉语编程等或汇编语言，而目标则是机器语言的目标代码，有时也称机器代码。

一句话描述编译与解释

编译：将一种语言（源码）转换成另一种语言（目标代码/计算机代码），发生在运行之前，产物是「另一份代码」，交给别人执行。

解释：把程序源代码一行一行的读懂然后执行，发生在运行时，产物是「运行结果」。



## 解释

解释是将程序代码逐行转换为机器代码并立即执行的过程。解释器是执行解释过程的工具，它可以直接读取源代码文件，逐行解释并执行代码。解释器可以在不需要编译的情况下执行代码，但解释的速度通常比编译的速度慢。

解释器：

直接把高级编程语言一行一行转译运行，重点是不会一次把整个程序转译出来，因此运行速度比较缓慢，它每转译一行程序就立刻运行，然后再转译下一行，再运行，如此不停地进行下去。

## 调试

调试是在程序运行时查找和修复错误的过程。调试器是用于执行调试过程的工具，它可以在程序运行时暂停执行、检查变量值、跟踪函数调用等。调试器可以帮助程序员定位并修复程序中的错误，提高程序的质量和可靠性。

<https://yncoders.com/show/73>

<https://www.bilibili.com/read/cv13258034/>

## 2、基本数据类型

## 3、程序基本语句

## 4、基本运算

## 5、数学库常用函数

```
#include <cmath>
```

**绝对值函数、四舍五入函数、下取整函数、上取整函数、平方根函数、常用三角函数、对数函数、指数函数**

### 绝对值函数 `abs(x)`

功能：返回参数 x 的绝对值。

```
int a = -5;
int b = abs(a); // b=5
double c = -5.2;
double d = abs(c) //d=5.2
```

### 四舍五入函数 `round(x)`

功能：返回参数 x 的四舍五入值。

```
double a = 3.4;
double b = 3.6;
int c = round(a); // c=3
int d = round(b); // d=4
```

### 下取整函数 `floor(x)`

功能：返回不大于参数 x 的最大整数。

```
double a = 3.4;
double b = 3.6;
int c = floor(a); // c=3
int d = floor(b); // d=3
```

### 上取整函数 `ceil(x)`

功能：返回不小于参数 x 的最小整数。

```
double a = 3.4;
double b = 3.6;
int c = ceil(a); // c=4
int d = ceil(b); // d=4
```

### 平方根函数 `sqrt(x)`

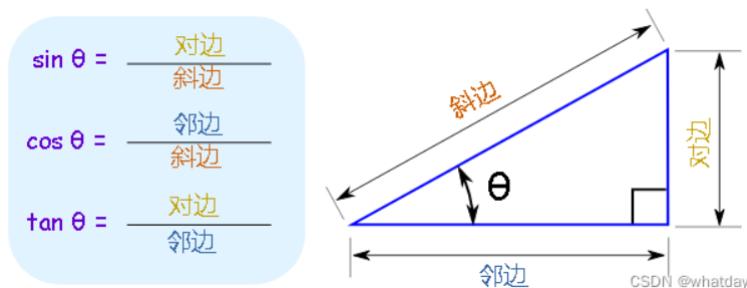
功能：表示求一个数的平方根。

```
double x = 4.0;
double y = sqrt(x); //计算 4 的平方根
```

## 常用三角函数

在直角三角形中，当平面上的三点A、B、C的连线，AB、AC、BC，构成一个直角三角形，其中 $\angle ACB$ 为直角。对 $\angle BAC$ 而言，对边(opposite)a=BC、斜边(hypotenuse)c=AB、邻边(adjacent)b=AC，则存在以下关系：

基本函数	英文	缩写	c++函数	表达式 ( $\theta$ 为 $\angle A$ 的角度 (弧度制))	语言描述
正弦函数	sine	sin	sin(x)	$\sin(\theta) = a/c$	$\angle A$ 的对边比斜边
余弦函数	cosine	cos	cos(x)	$\cos(\theta) = b/c$	$\angle A$ 的邻边比斜边
正切函数	tangent	tan	tan(x)	$\tan(\theta) = a/b$	$\angle A$ 的对边比邻边
余切函数	cotangent	cot	pow(tan(x), -1) or 1/tan(x)	$\cot(\theta) = b/a$	$\angle A$ 的邻边比对边
正割函数	secant	sec	pow(cos(x), -1) or 1/cos(x)	$\sec(\theta) = c/b$	$\angle A$ 的斜边比邻边
余割函数	cosecant	csc	pow(sin(x), -1) or 1/sin(x)	$\csc(\theta) = c/a$	$\angle A$ 的斜边比对边
反正弦函数	arcsine	arcsin	asin(x)	$\theta = \arcsin(a/c)$	求解给定正弦值的角度
反余弦函数	arccosine	arccos	acos(x)	$\theta = \arccos(b/c)$	求解给定余弦值的角度
反正切函数	arctangent	arctan	atan(x)	$\theta = \arctan(a/b)$	求解给定正切值的角度



```
double x = 1.0; //弧度制
double y = sin(x); // 计算 sin(x)
double z = tan(x); // 计算 tan(x)
```

## 对数函数

log : 自然对数

log10 : 以 10 为底的对数

log2 : 以 2 为底的对数

```
double x = 10.0;
double b = log(x); // 计算 ln(x)
double c = log10(x); // 计算 log10(x)
double d = log2(x); // 计算 log2(x)
```

## 指数函数

`exp(x)`：计算自然指数函数  $e^x$  的值 ( $e$  约等于 2.71828)。

`pow(base, exponent)`：计算 `base` 的 `exponent` 次幂的值。

`exp2(x)`：计算 2 的 `x` 次幂的值。

`pow(base, 0.5)`：计算 `base` 的平方根。

```
double x = 1.0;
double y = exp(x); // 计算  $e^x$ 
```

<https://blog.csdn.net/chaotiantian/article/details/115029466>

<https://baike.baidu.com/item/%E5%BC%A7%E5%BA%A6%E5%88%B6/3315973>

## 6、结构化程序设计

### 7、数组

### 8、字符串的处理

### 9、函数与递归

### 10、结构体与联合体

### 11、指针类型

### 12、文件及基本读写

### 13、STL 模板

栈 (stack)、队列 (queue)、链表 (list)、向量 (vector) 等容器

详细打开[C++提高编程.pdf](#)

## 2.1.3 数据结构

## 2.1.4 算法

### 1、算法概念与描述

算法描述：自然语言描述、流程图描述、伪代码描述

自然语言描述

通过自然语言来描述算法的步骤和操作。

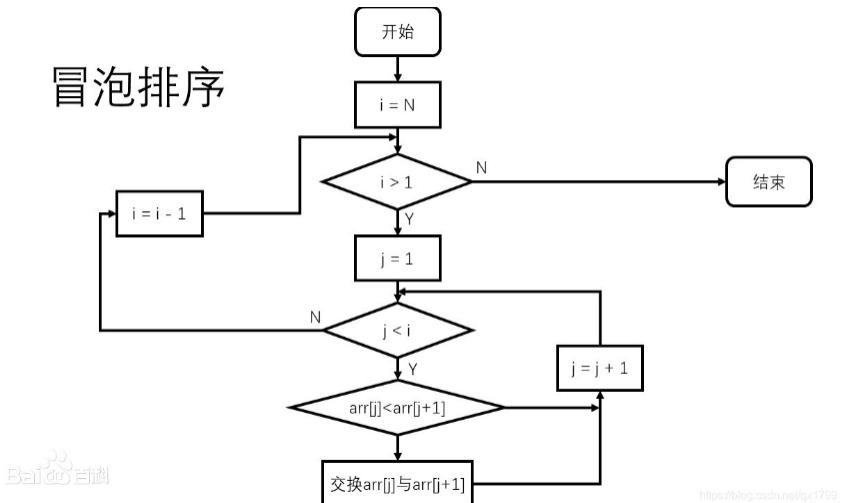
例如，冒泡排序算法的自然语言描述：

从数组的第一个元素开始，依次比较相邻的两个元素，如果前一个元素大于后一个元素，则交换它们的位置，直到将最大的元素移动到数组的最后一个位置。重复上述操作，直到所有元素都排好序为止。

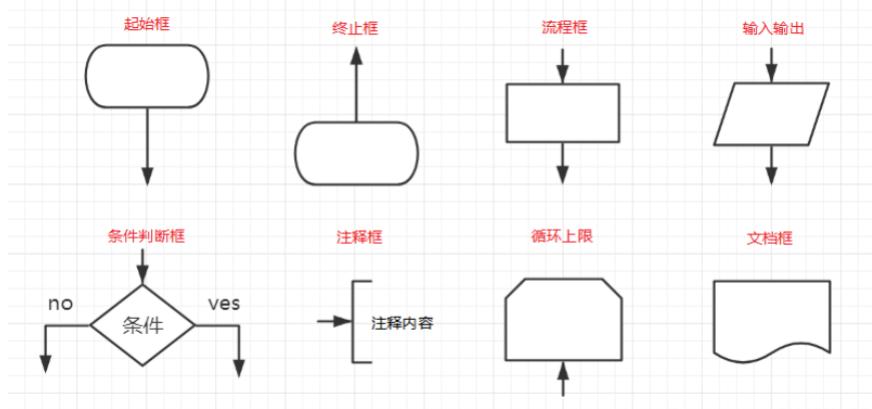
### 流程图描述

通过图形化方式来表示算法的步骤和操作。

例如，冒泡排序算法的流程图描述：



流程图中图形的表示含义



### 伪代码描述

通过一种类似编程语言的语法来描述算法的步骤和操作。伪代码通常比自然语言描述更具体和精确。

例如，冒泡排序算法的伪代码描述：

```

procedure bubbleSort(A : list of sortable items)
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n-1 do
            if A[i] > A[i+1] then
                swap(A[i], A[i+1])
                swapped = true
            end if
        end for
        n = n - 1
    until not swapped
end procedure

```

以上三种方式都可以用来描述算法，它们各有优缺点，选择哪种方式主要取决于具体情况和个人偏好。在实践中，通常使用**自然语言描述**和**伪代码**描述来表示算法。

## 2、入门算法

### 3、基础算法

#### 倍增法 (binary lifting)

介绍：倍增法 (binary lifting)，顾名思义就是翻倍。它能够使线性的处理转化为对数级的处理，大大地优化时间复杂度。这个方法在很多算法中均有应用，其中最常用的是 RMQ 问题和求 LCA (最近公共祖先)。倍增“与“二进制划分”两个思想相互结合，降低了求解很多问题的时间与空间复杂度。快速幂其实现是“倍增”与“二进制划分”思想的一种体现。其他应用还有，序列上的倍增问题，求解RMQ (区间最值) 问题的ST算法，求解最近公共祖先 (LCA) 等。

基本用法：

倍增主要用途是为了**查找单调数据组中某一数值**。

比如：在一个数组a {2,5,7,11,19} 中查找最大的小于12的数字。

- (1) 朴素做法：从第一个数开始，一个一个往后枚举，查找。
- (2) 二分做法：每次将数列分割一半判断，并且进一步查找子区间。
- (3) 倍增做法：设定一个增长长度 p 和已确定长度 l，现在要确定 a[l+p] 是否满足条件，若满足条件（比12小），则p成2倍增长；否则 p 缩小范围（试着缩小范围判断条件）。

```

l=0;
p=1;
while(p){//如果还能扩增范围(l)就继续
    if(a[l+p]<12){
        l+=p;//增加已知范围
        p<<=1//成倍增长，相当于p*=2
    }
    else{
        p>>=1;//缩小范围
    }
}
cout<<a[l];

```

倍增算法一般比较稳定，时间  $O(\log n)$ 。

## ST算法

ST算法(Sparse Table algorithm, ST)的基本步骤如下：

### 1、预处理：

创建一个二维数组 $dp$ ，其中 $dp[i][j]$ 表示从下标*i*开始长度为 $2^j$ 的区间内的最大值(或最小值)。

通过动态规划的方式计算 $dp$ 数组。对于任意*i*和*j*， $dp[i][j] = \max(dp[i][j-1], dp[i+2^{j-1}][j-1])$  (或最小值)。

这样可以在 $O(n \log n)$ 的时间内预处理完 $dp$ 数组。

### 2、查询：

给定一个查询区间 $[l, r]$ ，我们可以找到一个最大的*k*，使得 $2^k \leq r-l+1$ 。

此时，区间 $[l, r]$ 可以被分解为两个长度为 $2^k$ 的区间，即 $[l, l+2^k-1]$ 和 $[r-2^k+1, r]$ 。

查询这两个区间的最大值(或最小值)，取其中的较大值(或较小值)即可。这个过程只需 $O(1)$ 的时间复杂度。

## 最近公共祖先 (LCA)

最近公共祖先(Lowest Common Ancestor, LCA)

给定一颗有根树，若节点  $z$  既是节点  $x$  的祖先，也是节点  $y$  的祖先，则称  $z$  是  $x, y$  的公共祖先。在  $x, y$  的所有公共祖先中，深度最大的一个称为  $x, y$  的最近公共祖先，记为  $LCA(x, y)$ 。

$LCA(x, y)$  是  $x$  到根的路径与  $y$  到根的路径的交汇点，它也是  $x$  与  $y$  之间的路径上深度最小的节点。

这里着重介绍树上倍增法求LCA。

树上倍增法是一个很重要的算法。除了求 LCA 之外，它在很多问题中都有广泛应用。设  $F[x, k]$  表示  $x$  的  $2^k$  爺祖先，即从  $x$  向根节点走  $2^k$  步到达的节点。特别地，若该节点不存在，则令  $F[x, k] = 0$ 。  
 $F[x, 0]$  就是  $x$  的父节点。除此之外，任意  $k \in [1, \log n]$ ， $F[x, k] = F[F[x, k-1], k-1]$ 。

这类似于一个动态规划的过程，“阶段”就是节点的深度。因此，我们可以对树进行广度优先遍历，按照层次顺序，在节点入队之前，计算它在F数组中相应的值。

以上部分是预处理，时间复杂度为 $O(n \log n)$ ，之后可以多次对不同的  $x, y$  计算 LCA，每次询问的时间复杂度为  $O(\log n)$ 。

基于F数组计算  $LCA(x, y)$  分为以下几步：

设  $d[x]$  表示  $x$  的深度。不妨设  $d[x] \geq d[y]$  (否则交换  $x, y$ )

用二进制拆分思想，把  $x$  向上调整到与  $y$  同一深度。具体来说，就是依次尝试从  $x$  向上走  $k = 2^{\lceil \log n \rceil}, \dots, 2^1, 2^0$  步，检查到达的节点是否比  $y$  深。在每次检查中，若是，则令  $x = F[x, k]$

若此时  $x=y$ ，说明已经找到了 LCA，LCA 就等于  $y$ 。

用二进制拆分思想，把  $x, y$  同时向上调整，并保持深度一致且二者不相会。具体来说，就是依次尝试把  $x, y$  同时向上走  $k=2^{\lceil \log n \rceil}, \dots, 2^1, 2^0$  步，在每次尝试中，若  $F[x, k] \neq F[y, k]$  (即仍未相会)，则令  $x = F[x, k]$ ,  $y = F[y, k]$ 。

此时  $x, y$  必定只差一步就相会了，它们的父节点  $F[x, 0]$  就是 LCA。

```
#include<iostream>
#include<cmath>
#include<cstdio>
```

```

#include<cstring>
#include<algorithm>
using namespace std;

const int maxn=101000;
int n;
int m;
int s;
int tot;
int head[maxn];
int lg[maxn];
int depth[maxn];
int fa[maxn][32];

struct edge{
    int to;
    int from;
    int nxt;
}e[2*maxn];

void add(int x,int y){
    tot++;
    e[tot].to=y;
    e[tot].from=x;
    e[tot].nxt=head[x];
    head[x]=tot;
}

void dfs(int now,int fath){
    fa[now][0]=fath;
    depth[now]=depth[fath]+1;
    for(int i=1;i<=lg[depth[now]];i++) fa[now][i]=fa[fa[now][i-1]][i-1];
    for(int i=head[now];i;i=e[i].nxt){
        int y=e[i].to;
        if(y==fath) continue;
        dfs(y,now);
    }
}

int lca(int x,int y){
    if(depth[x]<depth[y]) swap(x,y);
    while(depth[x]>depth[y]) x=fa[x][lg[depth[x]-depth[y]]-1];
    if(x==y) return x;
    for(int k=lg[depth[x]]-1;k>=0;k--){
        if(fa[x][k]!=fa[y][k]){
            x=fa[x][k];
            y=fa[y][k];
        }
    }
    return fa[x][0];
}

int x,y;

int main(){

```

```

cin>>n>>m>>s;
for(int i=1;i<=n;i++){
    cin>>x>>y;
    add(x,y);
    add(y,x);
}
for(int i=1;i<=n;i++) lg[i]=lg[i-1]+(1<<lg[i-1]==i);
dfs(s,0);
for(int i=1;i<=m;i++){
    cin>>x>>y;
    cout<<lca(x,y)<<endl;
}
}

```

<https://www.dotcpp.com/course/947>

<https://www.luogu.com.cn/problem/P3379>

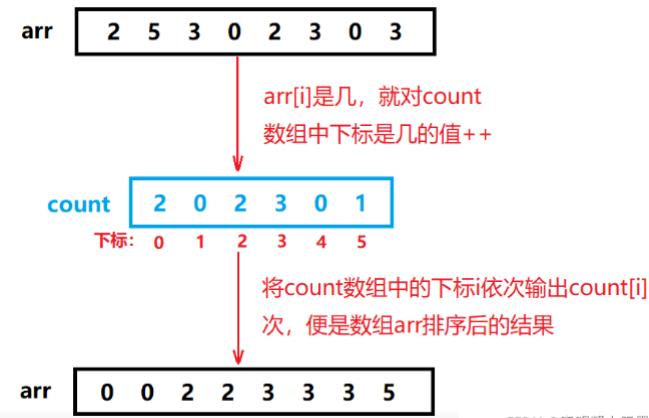
## 4、数值处理算法

### 5、排序算法

#### 计数排序

##### 概念

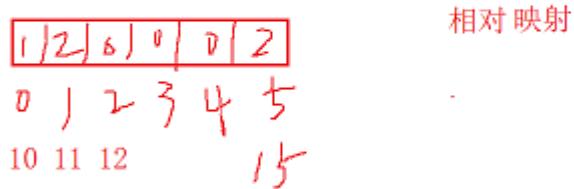
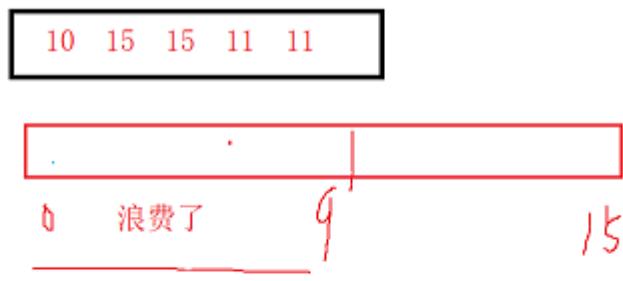
计数排序 (Counting sort) , 又叫非比较排序。顾名思义，该算法不是通过比较数据的大小来进行排序的，而是通过统计数组中相同元素出现的次数，然后通过统计的结果将序列回收到原来的序列中。



CSDN @森明帮大于黑虎帮

上列中的映射方法称为绝对映射，即arr数组中的元素是几就在count数组中下标为几的位置++，但这样会造成空间浪费。例如，我们要将数组：102,101,108，进行排序，难道我们要开辟109个整型空间吗？

若是使用计数排序，我们应该使用相对映射，简单来说，数组中的最小值就相对于count数组中的0下标，数组中的最大值就相对于count数组中的最后一个下标。这样，对于数组：102,101,108，我们就只需要开辟用于储存4个整型的空间大小了，此时count数组中下标为i的位置记录的实际上是 $101+i$ 这个数出现的次数。



10出现1次，把1次出现的0加min写回原数组

11出现两次，把两次出现的1加min写回两次回原数组

以此类推      相对映射

CSDN @森萌甜大于黑虎帮

## 总结

绝对映射：count数组中下标为i的位置记录的是arr数组中数字i出现的次数。

相对映射：count数组中下标为i的位置记录的是arr数组中数字min+i出现的次数。

计数排序是一种非基于比较的**非稳定线性排序算法**。

基本思想是：用空间换时间，本质上是建立了基于元素的Hash表。

## 代码

```

const int N = 100010;
const int W = 100010;

int n, w, a[N], cnt[w], b[N];

void counting_sort() {
    memset(cnt, 0, sizeof(cnt));
    for (int i = 1; i <= n; ++i) ++cnt[a[i]];
    for (int i = 1; i <= w; ++i) cnt[i] += cnt[i - 1];
    for (int i = n; i >= 1; --i) b[cnt[a[i]]--] = a[i];
}

```

时间复杂度： $O(N + \text{range})$  空间复杂度： $O(\text{range})$ 。

## 应用场景

计数排序只适用于数据范围较集中的序列的排序，若待排序列的数据较分散，则会造成空间浪费，并且计数排序只适用于整型排序，不适用与浮点型排序。

## 6、搜索算法

## 7、图论算法

### 泛洪算法 (flood fill)

#### 作用

用于确定连接到[多维数组](#)中给定节点的区域，可以用来标记或者分离图像的一部分，实现如Ps中自动选区功能。

#### 基本思想

顾名思义就像洪水漫过一样，把一块连通的区域填满。当然水要能漫过需要满足一定的条件，可以理解为满足条件的地方就是低洼的地方，水才能流过去。在图像处理中就是给定一个种子点作为起始点，向附近相邻的像素点扩散，把颜色相同或者相近的所有点都找出来，并填充上新的颜色，这些点形成一个连通的区域。

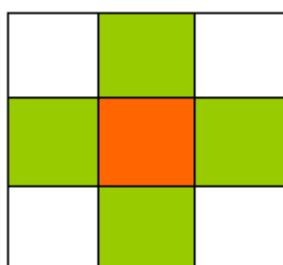
#### 算法参数

1. 起始节点 (start node)
2. 目标颜色 (target color)
3. 替换颜色 (replacement color)

#### 算法实现

漫水填充算法实现最常见有四邻域像素填充法，八邻域像素填充法，基于扫描线的填充方法。根据代码实现方式又可以分为递归与非递归。

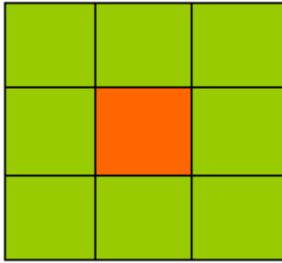
#### 四邻域递归实现



将像素点(x,y)周围的上下左右四个点分别进行着色。

```
void floodFill4(int x, int y, int newColor, int oldColor)
{
    if(x >= 0 && x < width && y >= 0 && y < height
        && getPixel(x, y) == oldColor && getPixel(x, y) != newColor)
    {
        setPixel(x, y, newColor); //set color before starting recursion
        floodFill4(x + 1, y, newColor, oldColor);
        floodFill4(x - 1, y, newColor, oldColor);
        floodFill4(x, y + 1, newColor, oldColor);
        floodFill4(x, y - 1, newColor, oldColor);
    }
}
```

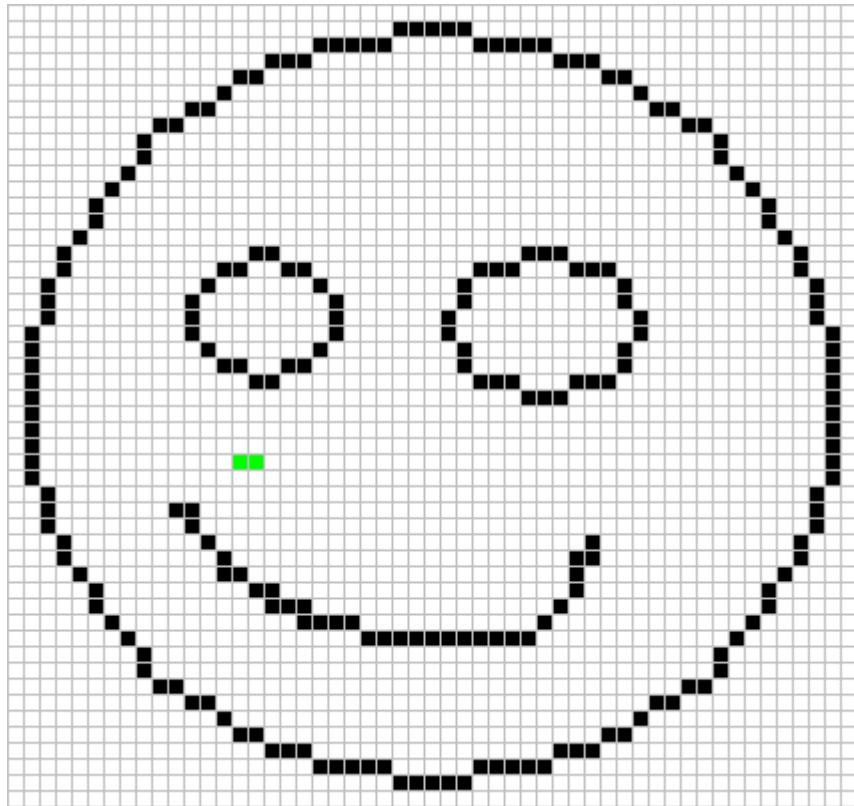
#### 八邻域递归实现



将一个像素点的上下左右，左上，左下，右上，右下都进行着色。

```
public void floodFill8(int x, int y, int newColor, int oldColor)
{
    if(x >= 0 && x < width && y >= 0 && y < height &&
       getPixel(x, y) == oldColor && getPixel(x, y) != newColor)
    {
        setPixel(x, y, newColor); //set color before starting recursion
        floodFill8(x + 1, y, newColor, oldColor);
        floodFill8(x - 1, y, newColor, oldColor);
        floodFill8(x, y + 1, newColor, oldColor);
        floodFill8(x, y - 1, newColor, oldColor);
        floodFill8(x + 1, y + 1, newColor, oldColor);
        floodFill8(x - 1, y - 1, newColor, oldColor);
        floodFill8(x - 1, y + 1, newColor, oldColor);
        floodFill8(x + 1, y - 1, newColor, oldColor);
    }
}
```

### 扫描线递归实现



先扫描一行或者一列内的连通像素，然后再上下行或者左右列扫描，可以减少递归栈的深度。

```

void floodFillScanline(int x, int y, int newColor, int oldColor){
    if(newColor==oldColor) return;
    if(screen[x][y]!=oldColor) return;
    int x1=x;
    while(x1<w&&screen[x1][y]==oldColor){
        screen[x1][y]=newColor;
        x1++;
    }
    x1=x-1;
    while(x1>=0&&screen[x1][y]==oldColor){
        screen[x1][y]=newColor;
        x1--;
    }
    x1=x;
    while(x1<w&&screen[x1][y]==newColor){
        if(y<h-1&&screen[x1][y+1]==oldColor)
            floodFillScanline(x1,y+1,newColor,oldColor);
        x1++;
    }
    x1=x-1;
    while(x1>0&&screen[x1][y]==newColor){
        if(y>0&&screen[x1][y-1]==oldColor)
            floodFillScanline(x1,y-1,newColor,oldColor);
        x1--;
    }
    x1=x;
    while(x1<w&&screen[x1][y]==newColor){
        if(y<h-1&&screen[x1][y-1]==oldColor)
            floodFillScanline(x1,y-1,newColor,oldColor);
        x1++;
    }
    x1=x-1;
    while(x1>0&&screen[x1][y]==newColor){
        if(y>0&&screen[x1][y-1]==oldColor)
            floodFillScanline(x1,y-1,newColor,oldColor);
        x1--;
    }
}

```

## 8、动态规划

### 简单区间类型动态规划

简单区间类型动态规划是指，**给定一个长度为 n 的数组，求其某个连续子数组的最大/最小值等类型的问题。**该问题可以使用动态规划来求解。

#### 例子

假设有一个数组 arr = {5, -6, 8, -7, 10, 2}，要求求出其所有子数组中的最大值。

动态规划的思路是，先定义一个 dp 数组，其中  $dp[i]$  表示以  $arr[i]$  结尾的子数组的最大值。那么对于  $dp[i]$ ，它可以由  $dp[i-1]$  和  $arr[i]$  计算得到。

具体地，考虑以  $arr[i]$  结尾的子数组的最大值，可能有两种情况：

- 以  $\text{arr}[i-1]$  结尾的最大子数组加上  $\text{arr}[i]$  仍然比  $\text{arr}[i]$  小，则以  $\text{arr}[i]$  结尾的最大子数组就是  $\text{arr}[i]$  本身；
- 以  $\text{arr}[i-1]$  结尾的最大子数组加上  $\text{arr}[i]$  比  $\text{arr}[i]$  大，则以  $\text{arr}[i]$  结尾的最大子数组就是以  $\text{arr}[i-1]$  结尾的最大子数组加上  $\text{arr}[i]$ 。

因此，可以得到状态转移方程为：

$\text{dp}[i] = \max(\text{dp}[i-1] + \text{arr}[i], \text{arr}[i]);$

最终的最大子数组和就是  $\text{dp}$  数组中的最大值。

## 代码

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

using namespace std;

int main() {
    vector<int> arr = {5, -6, 8, -7, 10, 2};
    int n = arr.size();

    vector<int> dp(n, 0);
    dp[0] = arr[0];
    int res = dp[0];

    for (int i = 1; i < n; i++) {
        dp[i] = max(dp[i-1] + arr[i], arr[i]);
        res = max(res, dp[i]);
    }

    cout << "The maximum sum of subarray is: " << res << endl;
}

return 0;
}
```

输出结果为：

```
The maximum sum of subarray is: 15
```

这说明，原数组  $\text{arr}$  的最大子数组和为 15，即  $\{8, -7, 10, 2\}$ 。

## 代码演示过程

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

int main() {
    // 输入数据
    int n;
    cin >> n;
    vector<int> a(n+1);
    for (int i = 1; i <= n; i++) {
```

```

    cin >> a[i];
}

// 初始化 dp 数组
vector<vector<int>> dp(n+1, vector<int>(n+1));
for (int i = 1; i <= n; i++) {
    dp[i][i] = a[i];
}

// 状态转移
for (int len = 2; len <= n; len++) {
    for (int i = 1; i <= n - len + 1; i++) {
        int j = i + len - 1;
        dp[i][j] = max(a[i] - dp[i+1][j], a[j] - dp[i][j-1]);
    }
}

// 输出结果
cout << dp[1][n] << endl;

return 0;
}

```

接下来用数据模拟展示：

假设输入数据为 5 2 7 8 1 2，那么初始化的 dp 数组为：

```

dp[1][1] = 2
dp[2][2] = 7
dp[3][3] = 8
dp[4][4] = 1
dp[5][5] = 2

```

接着，根据状态转移方程，依次计算 dp 数组的值：

```

dp[1][2] = max(2 - dp[2][2], 7 - dp[1][1]) = max(2 - 7, 7 - 2) = 5
dp[2][3] = max(7 - dp[3][3], 8 - dp[2][2]) = max(7 - 8, 8 - 7) = 1
dp[3][4] = max(8 - dp[4][4], 1 - dp[3][3]) = max(8 - 1, 1 - 8) = 7
dp[4][5] = max(1 - dp[5][5], 2 - dp[4][4]) = max(1 - 2, 2 - 1) = 1
dp[1][3] = max(2 - dp[2][3], 7 - dp[1][2]) = max(2 - 1, 7 - 5) = 2
dp[2][4] = max(7 - dp[3][4], 8 - dp[2][3]) = max(7 - 7, 8 - 1) = 7
dp[3][5] = max(8 - dp[4][5], 1 - dp[3][4]) = max(8 - 1, 1 - 7) = 7
dp[1][4] = max(2 - dp[2][4], 7 - dp[1][3]) = max(2 - 7, 7 - 2) = 5
dp[2][5]

```

## 2.1.5 数学与其他

### 1、数及其运算

## 2、初等数学

### 3、初等数论

#### 素数筛法：埃氏筛法与线性筛法

学习埃氏筛之前，我们先看一下暴力筛法，即对每个数都用试除法判断其是不是质数：

##### 暴力筛法

```
static final int N = 1e7 + 5;
int st[N]; // 初始化为0, 0表示质数, 1表示合数

for(int i = 2; i <= n; i++){
    for(int j = 2; j <= i / j; j++){//试除法
        if(i % j == 0){
            st[i] = 1; // 合数, 标记为1
        }
    }
}
```

##### 埃式筛

埃氏筛（埃拉托斯特尼筛法）。

思路：**一个质数的倍数一定是合数**，所以，假设P是质数，我们可以筛掉区间 [1,1e7] 中所有P的倍数。

先看个例子，对于数列1~11：

1 2 3 4 5 6 7 8 9 10 11

先筛去2的倍数：

1 2 3 4 5 6 7 8 9 10 11

[https://blog.csdn.net/god\\_0ne/](https://blog.csdn.net/god_0ne/)

然后筛去3的倍数：

1 2 3 4 5 6 7 8 9 10 11

[https://blog.csdn.net/god\\_0ne/](https://blog.csdn.net/god_0ne/)

然后筛去5的倍数：



[https://blog.csdn.net/qq\\_34010415/](https://blog.csdn.net/qq_34010415/)

至此，1~11内的所有合数都被筛完了，2 3 5 7 11是数列中的质数。为什么这样能筛去所有的合数呢，因为一个合数一定能被分解为几个质数的幂的乘积，并且这个数的质因子一定是小于它本身的，所以当我们从小到大将每个质数的倍数都筛去的话，当遍历到一个合数时，它一定已经被它的质因子给筛去了。

代码：

```
typedef long long LL;
const int N = 100000010;
int vis[N];//真为合数
int prim[N]; //记录质数
int cnt;//质数个数

void Eratosthenes(int n){//埃氏筛法
    for(LL i=2; i<=n; ++i){
        if( ! vis[i]){
            prim[++cnt] = i;
            for(LL j=i*i; j<=n; j+=i)
                vis[j] = 1;
        }
    }
}
//时间复杂度为: O(nloglog_2n)
```

模拟：

**n=20**  
i = 2: p{2}; v{4,6,8,10,12,14,16,18,20};  
i = 3: p{3}; v{9,12,15,18};  
i = 4:  
i = 5: p{5};  
i = 6:  
i = 7: p{7};  
i = 8,9,10:  
i=11: p{11};  
i=12:  
i=13: p{13};  
i = 14,15,16:  
i=17: p{17};  
i=18:  
i=19: p{19};  
i=20:

## 欧拉筛

欧拉筛法(线性筛法)

思路：

从小到大枚举每个数

1、如果当前数没划掉，必定是质数，记录该质数

2.枚举已记录的质数（如果合数已越界则中断）

(1)合数未越界，则划掉合数

(2)条件*i*%*p*==0，保证合数只被最小质因子划掉

若*i*是质数，则最多枚举到自身中断

若*i*是合数，则最多枚举到自身的最小质数中断

代码：

```
const int N = 100000010;
int vis[N]; //划掉合数
int prim[N]; //记录质数
int cnt; //质数个数

void get_prim(int n){ //线性筛法
    for( int i=2; i<=n; i++){
        if(!vis[i]) prim[++cnt] = i;
        for(int j=1; 111*i*prim[j]<=n; j++){
            vis[i*prim[j]] = 1;
            if(i%prim[j] == 0) break ;
        }
    }
} //时间复杂度O(n)
```

模拟：

**n=30**

i = 2: p{2}; v{4}; (2%2) break  
i = 3: p{3}; v{6,9}; (3%3) break  
i = 4: v{8}; (4%2) break  
i = 5: p{5}; v{10,15,25}; (5%5) break  
i = 6: v{12}; (6%2) break  
i = 7: p{7}; v{14,21}; (35>30)  
i = 8: v{16}; (8%2) break  
i = 9: v{18,27}; (9%3) break  
i=10: v{20}; (10%2) break  
i=11: p{11}; v{22}; (33>30)  
i=12: v{24}; (12%2) break  
.....  
i=16: (32>30)

[素数筛法详解：埃氏筛和欧拉筛-CSDN博客](#)

[G08 筛质数 埃氏筛法 线性筛法哔哩哔哩bilibili](#)

## 4、离散与组合数学

### 5、其他

#### 格雷码

在一组数的编码中，若任意两个相邻的代码只有一位二进制数不同，则称这种编码为格雷码（Gray Code），另外由于最大数与最小数之间也仅一位数不同，即“首尾相连”，因此又称循环码或反射码。

十进制数	自然二进制数	格雷码	十进制数	自然二进制数	格雷码
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

#### 认识格雷码

下为3位元的Gray Code：Gray Code是一个数列集合，每个数使用二进位来表示，假设使用n位元来表示每个数好了，任两个数之间只有一个位元值不同

000 001 011 010 110 111 101 100

#### 位元

位元就是数列的基础数是由几个二进位数来表示，就是几个位元。

上面的例子里面：000 是3个二进制数，那么就是3个位元，那么 $2^3 = 8$ ,总共8个位值

0000 0001 0011 0010 0110 0111 0101 0100 1100 1101 1111 1110 1010 1011 1001 1000

上面，0000 等是4个 二进制数，那么就是4个位元，那么 $4^3 = 16$ ,总共16个位值

#### 码值

001 就是一个码值

#### 格雷码的生成

第1步，改变最右边的位元值；

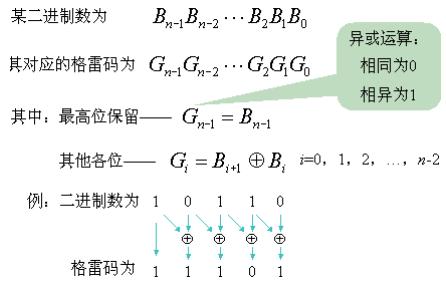
第2步，改变右起第一个为1的位元的左边位元；

重复第一步->第二步，直到所有的格雷码产生完毕。

#### 二进制转格雷码

方法1：

二进制码转换成二进制格雷码，其法则是保留二进制码的最高位作为格雷码的最高位，而次高位格雷码为二进制码的高位与次高位相异或，而格雷码其余各位与次高位的求法相类似。



方法2：

二进制最高位前补0，计算间隙左右两位异或值。（与方法1类似）

<http://t.csdnimg.cn/StSB9>

<http://t.csdnimg.cn/foHES>

<http://t.csdnimg.cn/h8KwE>

## 2.2 提高级

### 2.2.1 基础知识与编程环境

### 2.2.2 C++ 程序设计

#### 1、类 (class)

##### 类的概念及简单应用

##### 类的概念

(官方) 类是 C++ 的核心特性，在C++中，除了已有的数据类型（整型，字符型）外，还可以自定义数据类型，这种类型通常被称为用户定义的类型。类用于指定对象的形式，它包含了数据表示法（数据类）和用于处理数据（操作类）的方法。类中的数据和方法称为类的成员。函数在一个类中被称为类的成员。

(白话) 在C++中，用 "类" 来描述 "对象"，所谓的"对象"是指现实世界中的一切事物。那么类就可以看做是对相似事物的抽象，找到这些不同事物间的共同点，如自行车和摩托车，首先他们都属于"对象"，并且具有一定得相同点，和一些不同点，相同点如他们都有质量、都有两个轮子，都是属于交通工具等。"都有质量"、"两个轮子"属于这个对象的属性，而"都能够当做交通工具"属于该对象具有的行为，也称方法。

##### 简单应用

```
#include <iostream>
using namespace std;
class Box
{
public:
    double length;// 长度
    double breadth;// 宽度
    double height;// 高度
    void display()//打印对象的长宽高
    cout<<length<<endl<<breadth<<endl<<height<<endl;
    cout<<endl;
}
Box(double l,double b,double h){//普通含参构造函数
    length =l;
```

```

        breadth=b;
        height=h;
    }
    Box(Box &x){//自定义拷贝构造函数 复制一个现有的对象的数据
        length=x.length;
        breadth=x.breadth;
        height=x.height;
    }
};

int main()
{
    Box Box1(1,2,3);//声明 Box1, 类型为 Box 使用普通构造函数初始化
    Box Box2(Box1);//声明 Box2, 类型为 Box 使用拷贝构造函数初始化 复制Box1的数据内容
    Box1.display();//输出Box1内容
    Box2.display();//输出Box2内容
    return 0;
}

```

## 成员函数和运算符重载

详细打开[C++核心编程.pdf](#)

<http://t.csdnimg.cn/Vg2Pz>

<http://t.csdnimg.cn/nslzd>

## 2、STL 模板

容器 (container) 和迭代器 (iterator)

对 (pair) 、元组 (tuple)

集合 (set) 、多重集合 (multiset)

双端队列 (deque) 、优先队列 (priority\_queue)

映射 (map) 、多重映射 (multimap)

算法模板库中的常用函数

详细打开[C++提高编程.pdf](#)

### 2.2.3 数据结构

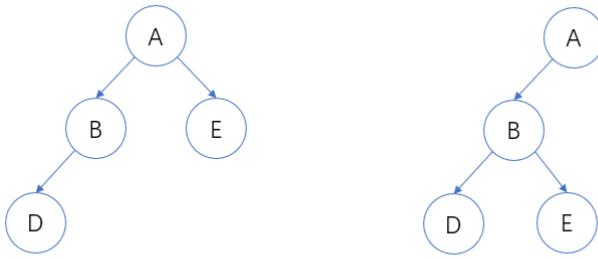
#### 1、线性结构

#### 2、集合与森林

树的孩子兄弟表示法

也叫树的二叉树表示法。树的左指针指向自己的第一个孩子，右指针指向与自己相邻的兄弟。

优点：它和二叉树的二叉链表表示完全一样。可利用二叉树的算法来实现对树的操作。



二叉树表示法  
<https://www.bing.net/cosdn.net/cjw838982809>

### 左孩子右兄弟表示的树的高度

因为二叉树表示法的根节点没有右孩子，所以树高就是左子树树高 + 1。

然后我们看下根节点第一个孩子的高度，由于第一个孩子的右子树和第一个孩子的高度是相同的，所以比较左子树 + 1的高度来和右子树来比较，如果 左子树 + 1 > 右子树，高度取左子树 + 1，否则取右子树。

```
//左孩子右兄弟表示的树的高度
int Height(Tree& t){
    if(t == NULL) return 0;
    return Height(t->left) + 1 > Height(t->right) ? Height(t->left) + 1 : Height(t->right);
}
```

### 左孩子右兄弟表示的树的叶子结点数

与求二叉树叶子结点的算法相似，仅需要把判断条件修改。

```
//左孩子右兄弟表示的树的叶子结点数
int Count(Tree& t){
    if(t == NULL) return 0;
    if(t->left == NULL) return 1;
    return Count(t->left) + Count(t->right);
}
```

### 树的总结点数

与求二叉树叶子结点的算法完全相同。

```
//总的结点数
int CountSum(Tree& t){
    if(t == NULL) return 0;
    return CountSum(t->left) + CountSum(t->right) + 1;
```

### 树的先根遍历

树的先根遍历，孩子是根，兄弟不是，多么精辟的话。先一边输出一边递归找孩子，等递归返回时再输出兄弟。

树的先根遍历的结果和二叉树的前序遍历的结果完全相同。

```
//树的先根遍历
void preOrder(Tree& t){
    if(t == NULL) return;
    cout<<t->val<<" ";
    TreeNode* p = t->left;
    while(p != NULL){
        preOrder(p); //递归找“根”
        p = p->right;
    }
}
```

## 树的后根遍历

在递归的边界处输出，先一直往下找“根”，找到最后走到孩子到了边界了，打印孩子之后再开始挖我的兄弟。

树的后根遍历的结果和二叉树的中序遍历的结果完全相同。

```
//树的后根遍历
void PostOrder(Tree& t){
    if(t == NULL) return;
    TreeNode* p = t->left;
    while(p != NULL){
        PostOrder(p); //递归找“根”
        p = p->right;
    }
    cout<<t->val<<" ";
}
```

## 左孩子右兄弟表示的树的层次遍历

每找到一个孩子时，迭代的把它的兄弟全部找出来，对每个结点都这样处理。

最后的结果按行来打印。

```
//左孩子右兄弟表示的树的层次遍历
void levelOrder(Tree& t){
    if(t == NULL) return;
    queue<TreeNode*> q;
    TreeNode* p;
    q.push(t);
    while(!q.empty()){
        int width = q.size();
        for(int i = 0;i < width;i++){
            p = q.front();
            q.pop();
            cout<<p->val<<" ";
            p = p->left;
            while(p != NULL){
                q.push(p);
                p = p->right;
            }
        }
        cout<<endl;
    }
}
```

```
    }  
}
```

## 左孩子右兄弟表示的树的宽度

与二叉树非递归找宽度的原理完全相同，既然可以按层打印，就可以比较记录最大的宽度。

```
//左孩子右兄弟表示的树的宽度  
int width(Tree& t){  
    if(t == NULL) return 0;  
    queue<TreeNode*> q;  
    TreeNode* p;  
    int max = 0;  
    q.push(t);  
    while(!q.empty()){  
        int width = q.size();  
        for(int i = 0;i < width;i ++){  
            p = q.front();  
            q.pop();  
            p = p->left;  
            while(p != NULL){  
                q.push(p);  
                p = p->right;  
            }  
        }  
        max = max < width ? width : max;  
    }  
    return max;  
}
```

## 在以t为根的树中找结点p的双亲

循长子的兄弟链，递归在子树中搜索。

```
//在以t为根的树中找结点p的双亲  
TreeNode* findParent(Tree& t,TreeNode* p){  
    if(t == NULL || p ==NULL) return NULL;  
    TreeNode* q = t->left;  
    TreeNode* s;  
    //循长子的兄弟链，递归在子树中搜索  
    while(q != NULL && q != p){  
        if(s = findParent(p,q) != NULL) return s; //找到双亲，返回  
        q = q->right;  
    }  
    if(q != NULL && q == p) return t; //找到双亲  
    else return NULL; //未找到  
}
```

树采用先序构造，大家可以自己画一下。

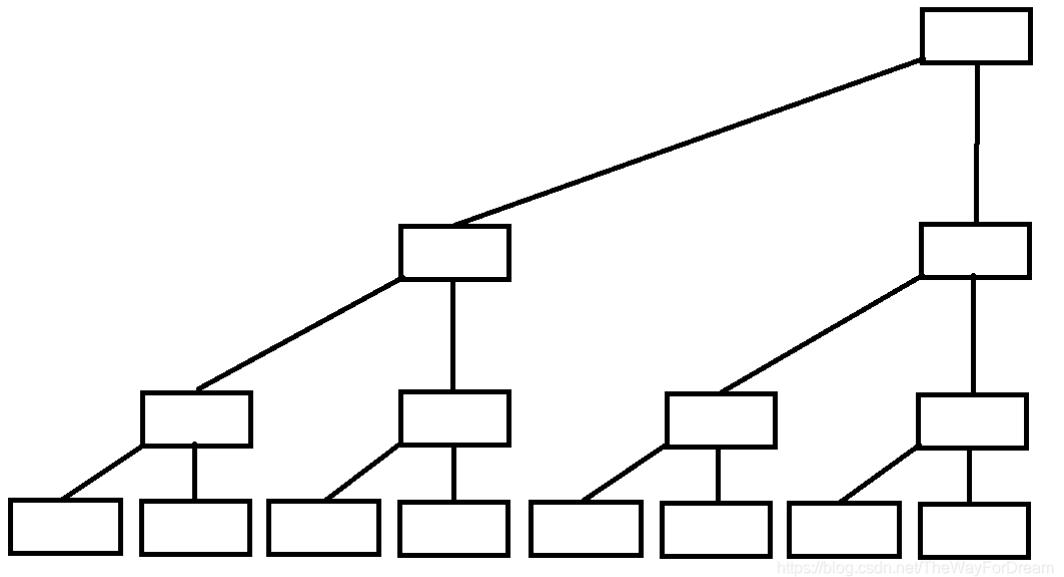
[完整代码](#)

<http://t.csdnimg.cn/Ea86n>

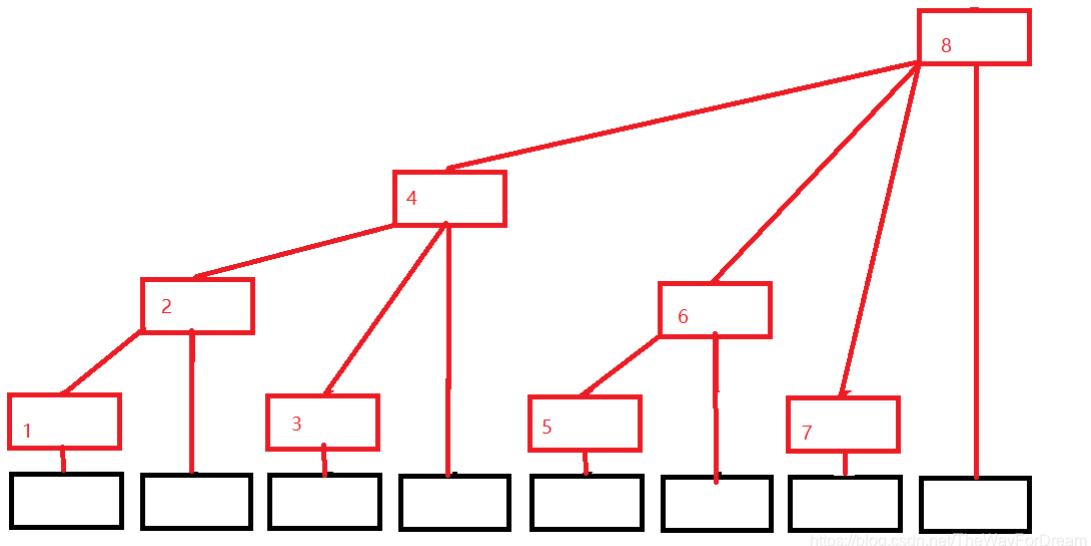
### 3、特殊树

#### 树状数组

这是二叉树的结构



这是树状数组的结构



#### 概念

树状数组是一种支持 **单点修改** 和 **区间查询** 的，代码量小的数据结构。

#### 前置知识

$lowbit(x)$  运算

如何计算一个非负整数  $n$  在二进制下的最低为1及其后面的0构成的数？

例如： $44 = (101100)_2$ , 最低为1和后面的0构成的数是  $(100)_2 = 4$  所以

$lowbit(44) = lowbit((101100)_2) = (100)_2 = 4$ , 那么  $lowbit$  运算时怎么实现的呢？

44的二进制=(101100), 我们对44的二进制数取反+1, 也即~44+1, 得到-44

-44的二进制=(010100), 然后我们把44和-44的二进制进行按位与(&)运算, 得到  $(000100)_2 = (4)_{10}$

所以 $lowbit(x) = x \& (-x)$

## 问题引入

给出一个长度为  $n$  的数组, 完成以下两种操作

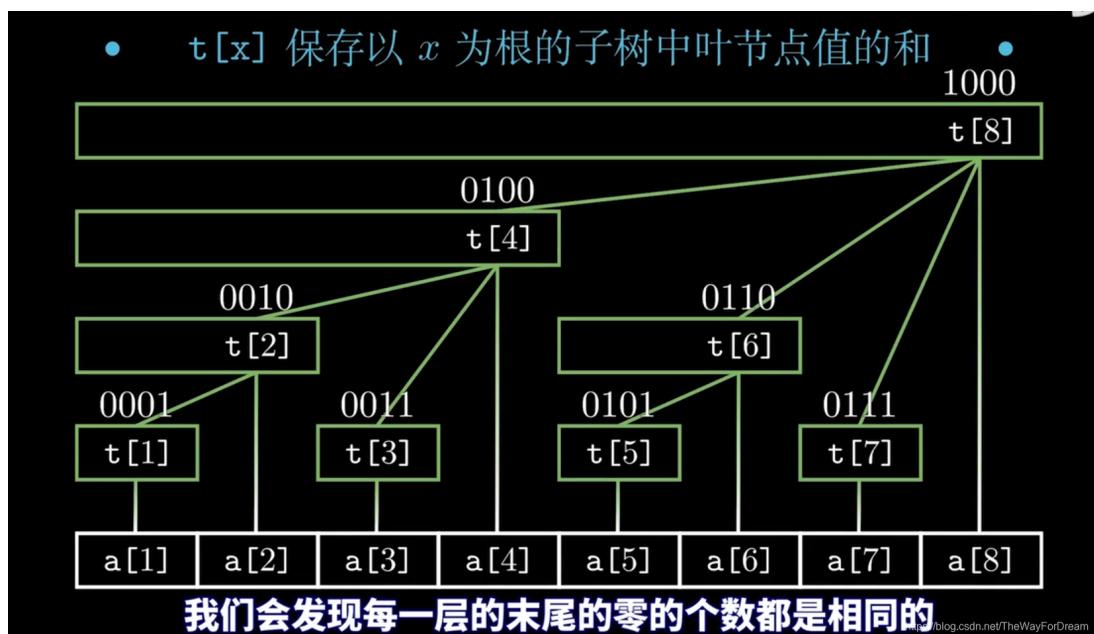
- 将第  $x$  个数加上  $k$
- 输出区间  $[x, y]$  内每个数的和

<https://blog.csdn.net/TheWayForDream>

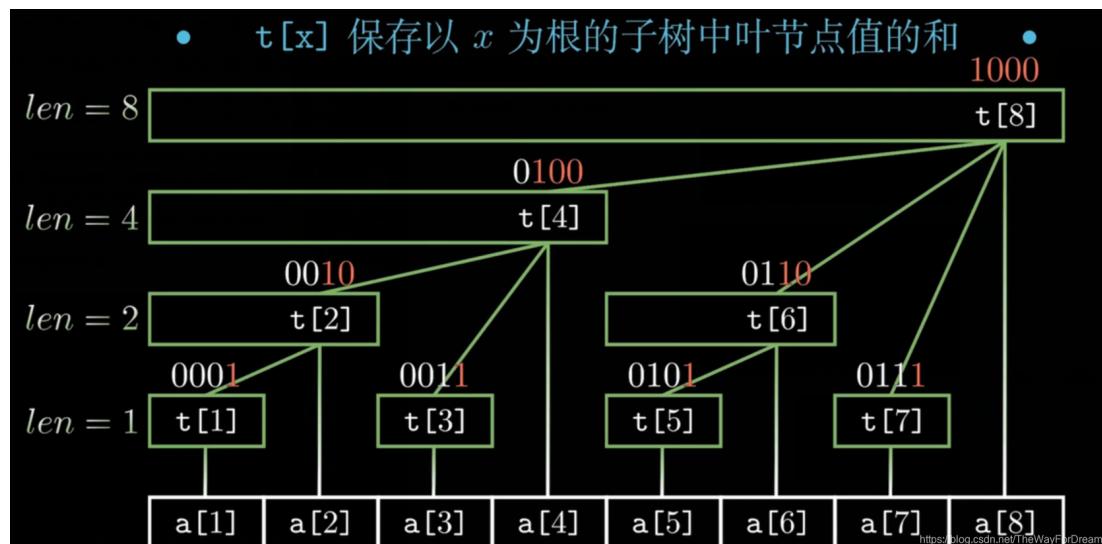
显然, 我们一开始会想到暴力的朴素做法, 单点修改操作时间复杂度  $O(1)$ , 区间求和, 暴力遍历区间每一个数再相加时间复杂度  $O(n)$ , 如果区间求和查询的次数为  $n$  次, 那么中的时间复杂度为  $O(n^2)$ , 对于大数据的题来说肯定会 T, 此时如果用树状数组的话复杂度可以讲到  $O(n \log n)$ .

## 树状数组结构分析

接下来分析树状数组的原理



上面是树状数组的结构图,  $t[x]$  保存以  $x$  为根的子树中叶子节点值的和, 原数组为  $a[]$ 。那么原数组前 4 项的和  $t[4] = t[2] + t[3] + a[4] = t[1] + a[2] + t[3] + a[4] = a[1] + a[2] + a[3] + a[4]$ , 看似没有什么特点, 别着急往下看



我们通过观察节点的二进制数，进一步发现，树状数组中节点 $x$ 的父节点为 $x + \text{lowbit}(x)$ ,例如 $t[2]$ 的父节点为 $t[4] = t[2 + \text{lowbit}(2)]$

### 单点修改，区间查询

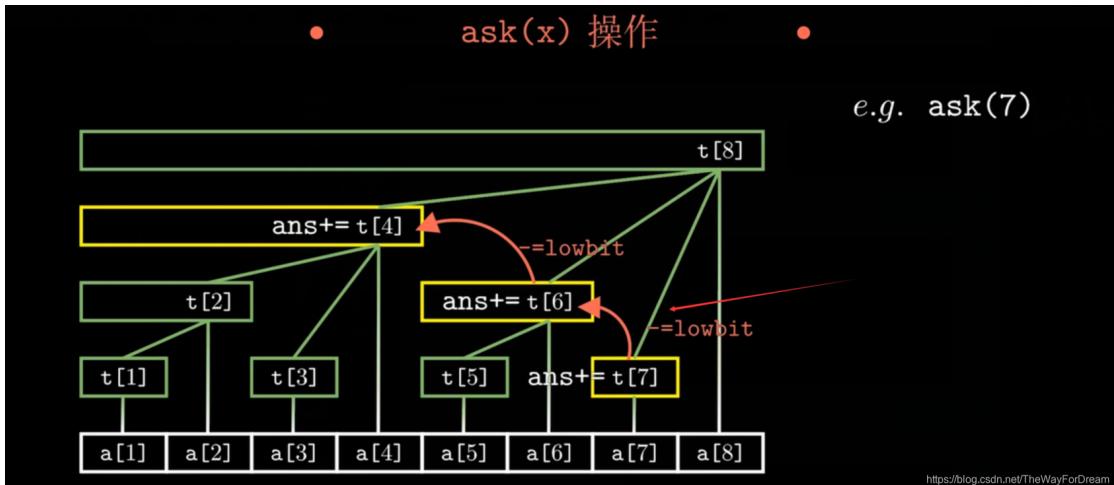
所以我们在**单点修改**的同时，更新父节点就变得尤为简单，例如我们对 $a[1] + k$ ，那么祖先节点 $t[1], t[2], t[4], t[8]$ 都需要 $+k$ 更新(因为 $t[]$ 表示前缀和)，此时我们就可以用 $+\text{lowbit}$ 操作实现。

### 代码

```
int add_dandian(int x, int k)
{
    for(int i=x; i<=n; i+=lowbit(i))
        t[i] += k;
}
```

那么单点修改实现了，如何实现区间查询呢？

例如：我们需要查询前7项的区间和 $\text{sum}[7]$



通过图中不难看出， $\text{sum}[7] = t[7] + t[6] + t[4]$ ，我们进一步发现， $6 = 7 - \text{lowbit}(7)$ ,  $4 = 6 - \text{lowbit}(6)$ ，所以我们可以通过不断的 $-\text{lowbit}$ 操作来实现求和

```
int ask(x){
    int sum = 0;
    for(int i=x; i>=1; i-=lowbit(i)){
        sum += t[i];
    }
    return sum;
}
```

这只能求区间 $[1, x]$ 的区间和，那么如何求 $[L, R]$ 的区间和呢？这时候利用前缀和相减的性质就可以了， $[L, R] = [1, R] - [1, L - 1]$

### 代码

```

int search(int L, int R)
{
    int ans = 0;
    for(int i=L-1; i;i-=lowbit(i))
        ans-=c[i];
    for(int i=R; i;i-=lowbit(i))
        ans+=c[i];
    return 0;
}

```

## 区间修改，单点查询

对于这一类操作，我们需要构造出原数组的差分数组 $b$ ，然后用树状数组维护 $b$ 数组即可。对于区间修改的话，我们只需要对差分数组进行操作即可，例如对区间 $[L, R] + k$ ，那么我们只需要更新差分数组 $add(L, k), add(R + 1, -k)$ ，这是差分数组的性质。

### 代码

```

int update(int pos, int k)//pos表示修改点的位置，K表示修改的值也即+K操作
{
    for(int i=pos; i<=n; i+=lowbit(i))
        c[i]+=k;
    return 0;
}
update(L,k);
update(R+1,-k);

```

对于单点查询操作，求出 $b$ 数组的前缀和即可，因为 $a[x] = \text{差分数组 } b[1] + b[2] + \dots + b[x]$ 的前缀和，这是差分数组的性质之一。

### 代码

```

11 ask(int pos)//返回区间pos到1的总和
{
    11 ans=0;
    for(int i=pos; i;i-=lowbit(i)) ans+=c[i];
    return ans;
}

```

<http://t.csdnimg.cn/zTpCy>

[树状数组 - OI Wiki \(oi-wiki.org\)](#)

[差分数组详解](#)

[P3374 【模板】树状数组 1 - 洛谷 | 计算机科学教育新生态 \(luogu.com.cn\)](#)

[P3368 【模板】树状数组 2 - 洛谷 | 计算机科学教育新生态 \(luogu.com.cn\)](#)

## 笛卡尔树

## 定义

笛卡尔树是一种特定的**二叉树**数据结构，可由数列构造，在范围最值查询、范围top k查询(range ,top k queries)等问题上有广泛应用。它具有堆的有序性，中序遍历可以输出原数列。

## 性质

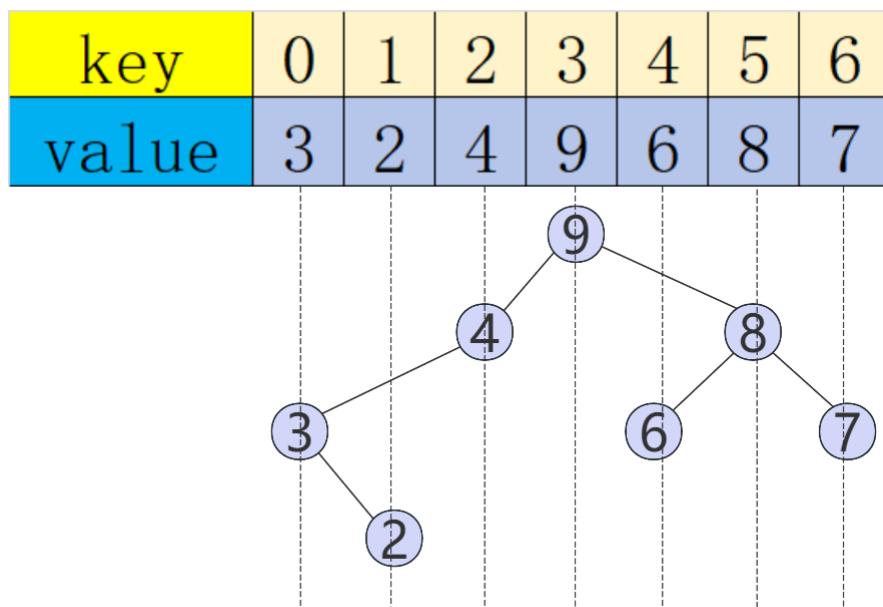
无相同元素的数列构造出的笛卡尔树具有下列性质：

- 1.结点——对应于数列元素。即数列中的每个元素都对应于树中某个唯一结点，树结点也对应于数列中的某个唯一元素
- 2.中序遍历(in-order traverse)笛卡尔树即可得到原数列。（即任意树结点的左子树结点所对应的数列元素下标比该结点所对应元素的下标小，右子树结点所对应数列元素下标比该结点所对应元素下标大。）
- 3.树结构存在堆序性质，即任意树结点所对应数值大/小于其左、右子树内任意结点对应数值

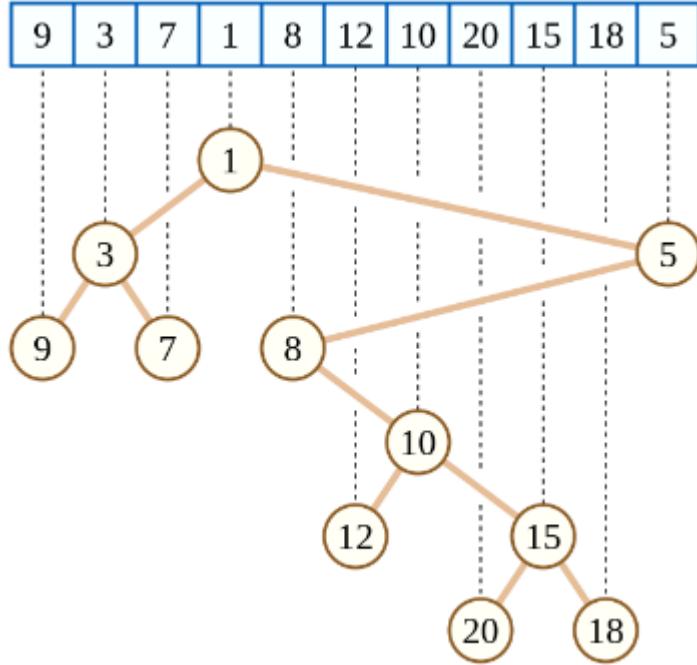
根据堆序性质，笛卡尔树根结点为数列中的最大/小值，树本身也可以通过这一性质递归地定义：根结点为序列的最大/小值，左、右子树则对应于左右两个子序列，其结点同样为两个子序列的最大/小值。因此，上述三条性质唯一地定义了笛卡尔树。

## 例子

构造具有大根堆性质的笛卡尔树：



构造具有小根堆性质的笛卡尔树：



## 代码

```

int st[N], ls[N], rs[N], n, A[N]; // ls代表笛卡尔树每个节点的左孩子，rs代表笛卡尔树每个节点的右孩子
int top = 0;
//构造小根堆性质的笛卡尔树
for (int i = 1; i <= n; ++i)
{
    while (top && A[st[top]] > A[i]) ls[i] = st[top--]; //栈顶元素为当前元素的左孩子
    if (top) rs[st[top]] = i; //当前元素为栈顶元素的右孩子
    st[++top] = i;
}

```

[数据结构——二叉搜索树详解-CSDN博客](#)

[P5854 【模板】笛卡尔树 - 洛谷 | 计算机科学教育新生态\(luogu.com.cn\)](#)

[神秘数据结构：笛卡尔树 - Flandre-Zhu - 博客园\(cnblogs.com\)](#)

<https://oi-wiki.org/ds/cartesian-tree/>

[https://www.bilibili.com/video/BV1mE421G7jk/?share\\_source=copy\\_web&vd\\_source=97161cd20dc703f41cf60bcb19a74f71](https://www.bilibili.com/video/BV1mE421G7jk/?share_source=copy_web&vd_source=97161cd20dc703f41cf60bcb19a74f71)

平衡树：AVL、treap、splay等

平衡二叉树（AVL）

## 概念

平衡二叉树全称叫做平衡二叉搜索（排序）树，简称AVL（是大学教授 G.M. Adelson-Velsky 和 E.M. Landis 名称的缩写）树。

## 性质

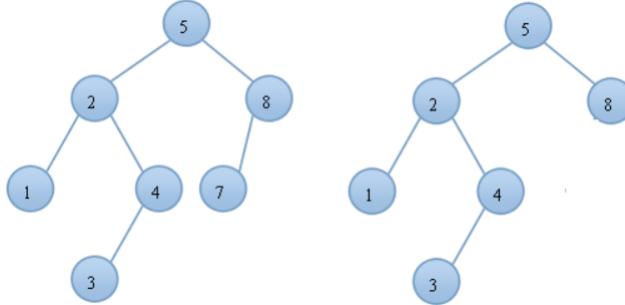
- 是一颗二叉搜索树（Binary Search Tree, BST）
- 它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，

- 左右两个子树也都是一棵平衡二叉树。

### 平衡因子 (Balance Factor, bf)

$bf = \text{左子树的深度} - \text{右子树的深度}$ 。 (在 AVL树中，必须满足:  $-1 \leq bf \leq 1$ )

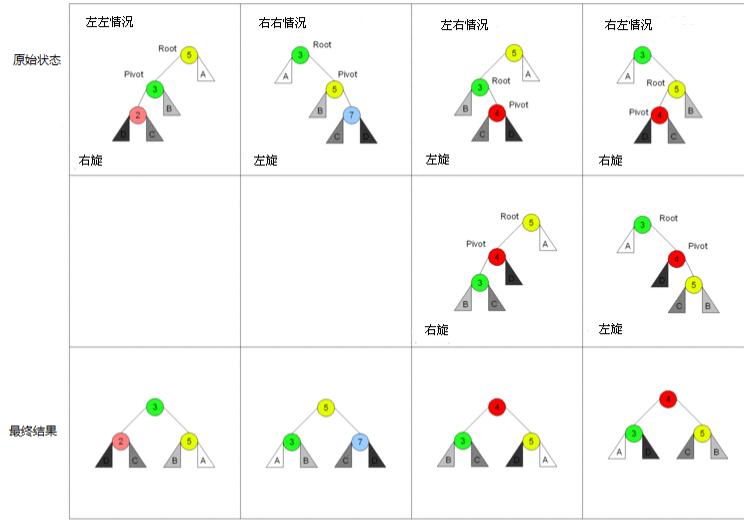
### 区分



在插入的过程中，会出现一下四种情况破坏AVL树的特性，我们可以采取如下相应的旋转。

插入位置	状态	操作
在结点T的左结点 (L) 的 <b>左子树 (L)</b> 上做了插入元素	左左型	右旋
在结点T的左结点 (L) 的 <b>右子树 (R)</b> 上做了插入元素	左右型	左右旋
在结点T的右结点 (R) 的 <b>右子树 (R)</b> 上做了插入元素	右右型	左旋
在结点T的右结点 (R) 的 <b>左子树 (L)</b> 上做了插入元素	右左型	右左旋

注意： $T$  表示失衡节点。

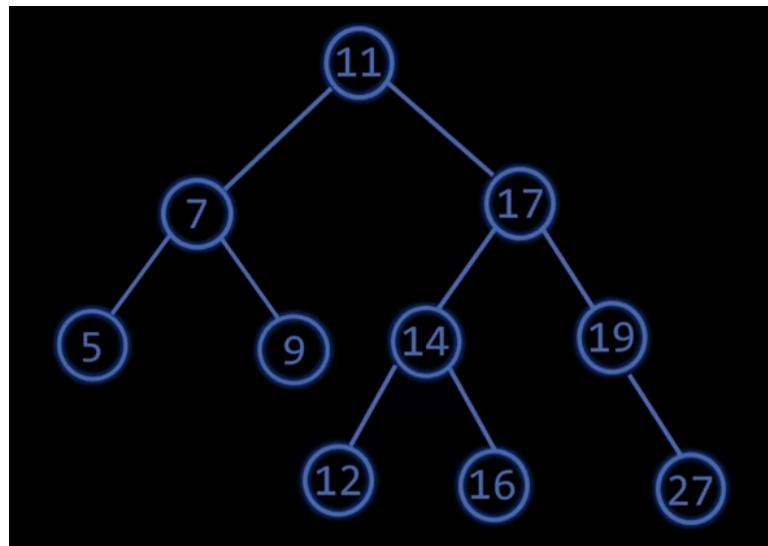


来自于维基百科

### 构建AVL树

每次插入需要检查各祖先平衡因子，找到最近失衡并调整。

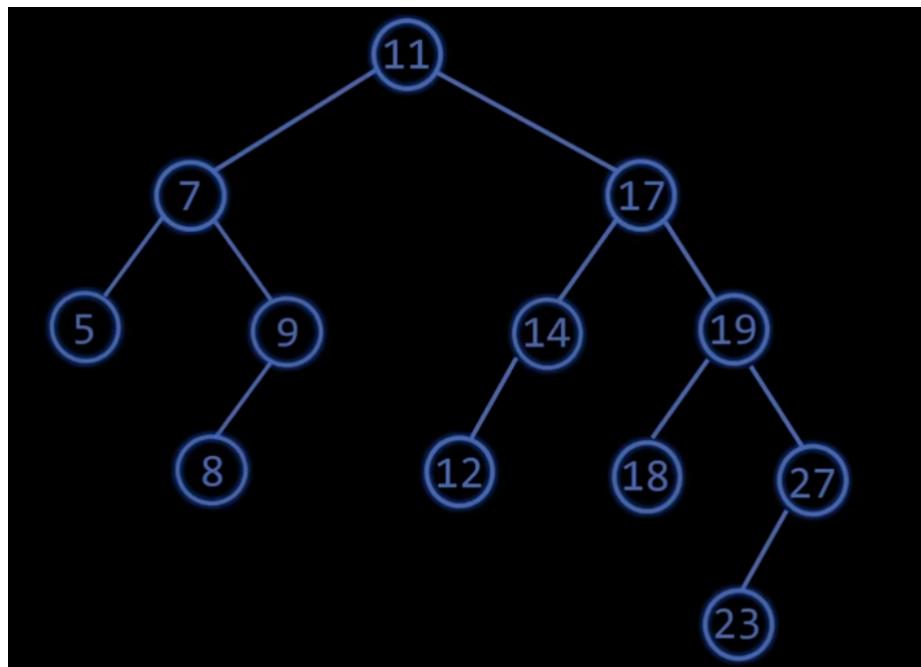
例如：14 9 5 17 11 12 7 19 16 27



## 删除AVL树

删除结点后需要依次对每个祖先检查并调整。

例如：删除5



<https://www.bilibili.com/video/BV1tZ421q72h>

<https://blog.csdn.net/xiaojin21cen/article/details/97602146>

<https://www.cnblogs.com/idreamo/p/8308336.html>

<https://segmentfault.com/a/1190000006123188>

[https://blog.csdn.net/sjg\\_sjk/article/details/80332151](https://blog.csdn.net/sjg_sjk/article/details/80332151)

<https://blog.csdn.net/FreeeLinux/article/details/52204851>

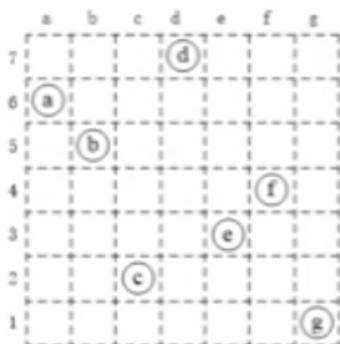
## Treap树 (树堆)

Treap =Tree + Heap, 树和堆的结合

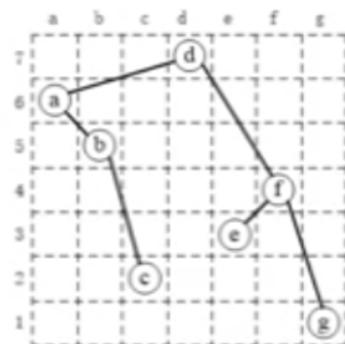
- 每个结点有2个值：键值、优先级。
- 对于键值来说，这棵树是**BST**；对于优先级来说，这棵树是一个**堆**。
- Treap树有唯一的形态。解释：令每个结点的优先级互不相等，那么整棵树的形态是唯一的，和元素的插入顺序没有关系。

### • 例如：

- 键值：{a, b, c, d, e, f, g}
- 优先级：{6, 5, 2, 7, 3, 4, 1}。



(1)键值和优先级



(2)建树



(3)形成的Treap树

## Treap树如何解决平衡问题？

- 合理分配结点的优先级，可以得到一个比较平衡的BST。
- 一个简单的分配方法：随机。对每个结点的优先级进行随机赋值，生成的Treap树的形态也是随机的。
- 虽然不能保证每次生成的Treap树是平衡的，但是期望的插入、删除、查找的时间复杂度都是 $O(\log n)$ 的。

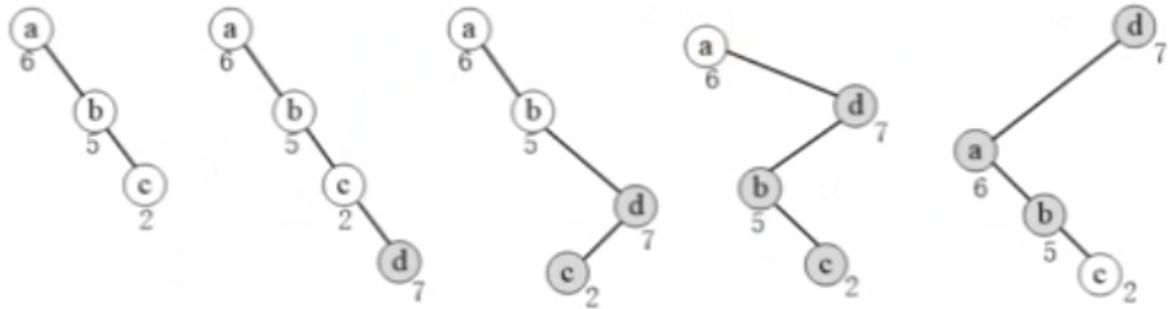
## 插入

把新结点x插入到Treap树，分两步：

- 先把x按键值大小插入到合适的子树上。
- 给x随机分配一个优先级，如果x的优先级违反了堆的性质，即它的优先级比父结点高，那么进行调整，让x往上走，替代父结点，最后得到一个新的Treap树。

例：新结点的插入和调整

1. 图 (2) 插入d点，按朴素的插入方法插入到底部；
2. 图 (3) d的优先级比父结点c高，左旋，上升；
3. 图 (4) d的优先级比新的父结点b高，继续左旋上升；
4. 图 (5) 再次左旋上升，完成了新的Treap树。



(1) 初始态 (2) 插入d (3) d左旋 (4) d左旋 (5) d左旋

### 删除

- 待删除的结点x是叶子结点：直接删除。
- 待删除的结点x有子结点：找到优先级最大的子结点，把x向相反的方向旋转，也就是把x向树的下层调整，直到x被旋转到叶子结点，然后直接删除。

### 分裂与合并

- 把一棵树分裂成两棵树，或者把两棵树合并成一棵。Treap树做这样的操作，**比较繁琐**。
- 一般用 splay树 做分裂与合并。

<https://www.bilibili.com/video/BV1hh4y1W7Zd>

<https://oi-wiki.org/ds/treap/>

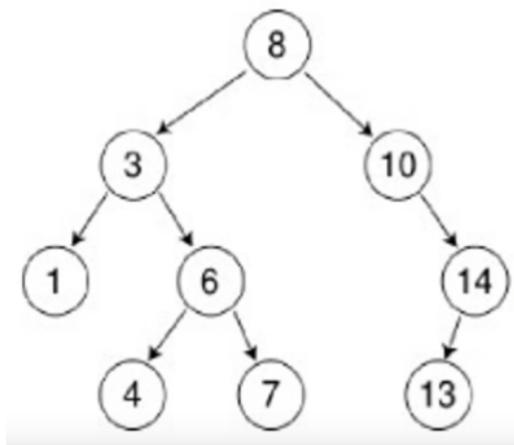
<https://blog.csdn.net/yandaoqiuusheng/article/details/85037548>

### 伸展树 (Splay)

#### 概念

Splay树（伸展树）是BST，它的查找、插入、删除、分割、合并等操作，复杂度都是 $O(\log n)$ 的。最大的特点：可以把某个结点往上旋转到指定位置，特别是可以旋转到根的位置，成为新的根结点。一个应用：如果需要经常查询和使用一个数，那么把它旋转到根结点，下次访问它，只需要查一次就找到了。

例如：频繁查找5



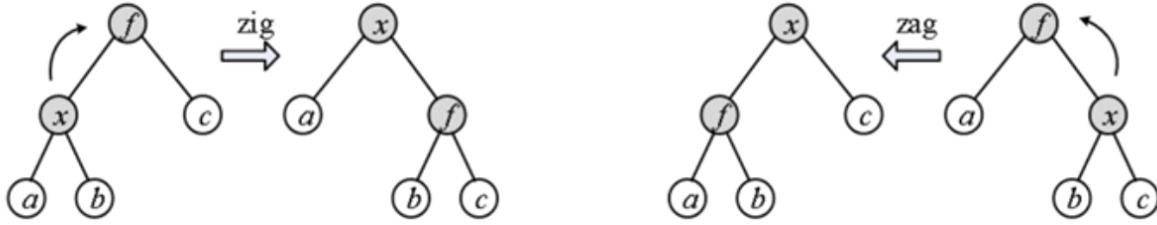
#### 核心

把结点旋转到根(提根)

#### 操作

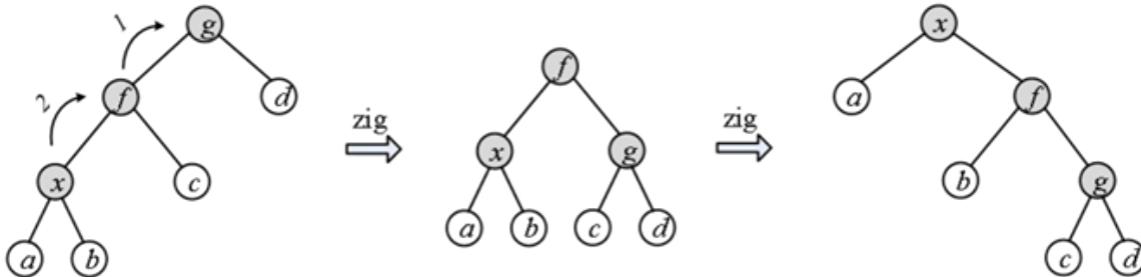
### (1) 单旋

做一次zig或zag。此时x已经旋转上升到距离根只有一层的位置，父结点f就是根，只需要做一次单旋即可。  
(和Treap树中的旋转法完全一样)



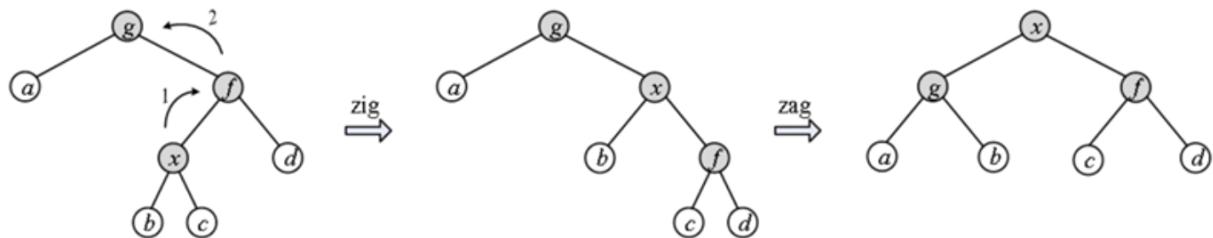
### (2) 一字旋

x、f、g在一条线上，做双旋zig-zig或zag-zag。若x是f的左儿子，f是g的左儿子，做两次zig；若都是右儿子，做两次zag。



### (3) 之字旋

x、f、g不在一条线上，做双旋zig-zag或zag-zig。若x是f的左儿子，f是g的右儿子，先zig再zag；否则先zag再zig。



### 旋转的复杂度

- 旋转一次的时间是个常数。
- 把x从所在的深度提到根，总复杂度是多少？
  - 如果是平衡二叉树，最深的结点深度是 $O(\log n)$ ，总复杂度就是 $O(\log n)$ 。
  - 在**均摊**意义上，可以把Splay提根操作的复杂度看成是 $O(\log n)$ 。

### 功能

#### (1) 旋转

`splay(x, p)`, 把结点x旋转到位置p。

- $p = 0$ 表示把x旋到根，x是整棵树的根；
- $p \neq 0$ , 表示把x旋到p的子树上，p成为x的父亲。

#### (2) 分裂

`split(p, &L, &R)`, p是排名或键值, 按p把树分裂为两棵树L、R。例如p是排名, 把第p个结点x旋到根, 然后断开x和右子树, 以x为根的是L, 右子树是R。

### (3) 合并

`Merge(L,R)`, 合并两棵树L、R。**缺省条件是L的所有结点比R小**。首先把R的最小结点x旋到根, 此时x没有左儿子, 然后让L做x的左儿子, 就合并了两棵树。也可以把L的最大结点旋到根, 后续操作类似。

### (4) 插入

`Insert()`, 先把新结点按BST的规则插入到一个空的叶子结点上, 然后旋到根。

### (5) 删除

`del()`, 把要删除的结点旋到根, 删掉它, 然后合并它的左右儿子。

## 代码

```
void Rotate(int x){  
    int y = fa[x], z = fa[y]; //y为x的父结点, z为x的祖父结点  
    int b = (lc[y] == x) ? rc[x]:lc[x];  
    //选取不以y->x同侧的x的儿子b, 以维持BST性质  
    fa[x] = z, fa[y] = x; //将x的父结点接到z上, y的父结点接到x  
    if(b) fa[b] = y; //若b为空则不能将父结点接到y上  
    if(z) (y == lc[z] ? lc[z] : rc[z]) = x; //以x接替y的位置  
    if (x == lc[y]) rc[x] = y, lc[y] = b; //调整各结点之间的父子关系  
    else lc[x] = y, rc[y] = b;  
}
```

<https://www.bilibili.com/video/BV1gm4y1U7Hq>

<https://www.bilibili.com/video/BV1BG4y1M7dB>

[数据结构\]伸展树\(Splay Tree\)原理及若干应用详解 \(无指针\)\\_splay tree 应用-CSDN博客](#)

## 4、常见图

### 偶图 (二分图) (bipartite graph)

#### 偶图的定义

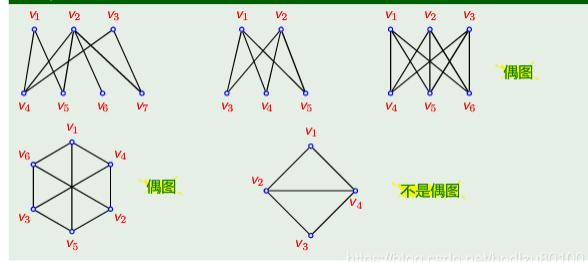
考虑:有一组工人和一批工作任务作为图中的结点,并根据工人对任务是否熟悉来建立边的连接。在这样的图中,工人之间没有边,工作任务之间也不会有边,所有的边都存在于工人组和任务组之间。这样的图称为偶图。

#### Definition

若无向图  $G = \langle V, E \rangle$  的结点集  $V$  能够划分为两个子集  $V_1, V_2$ , 满足  $V_1 \cap V_2 = \emptyset$ , 且  $V_1 \cup V_2 = V$ , 使得  $G$  中任意一条边的两个端点, 一个属于  $V_1$ , 另一个属于  $V_2$ , 则称  $G$  为偶图(bipartite graph)或二分图或二部图。 $V_1$  和  $V_2$  称为互补结点子集, 偶图通常记为  $G = \langle V_1, E, V_2 \rangle$ 。

<https://blog.csdn.net/hpdizu80100>

#### Example



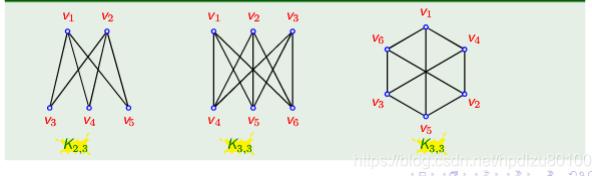
<https://blog.csdn.net/hpdizu80100>

## 完全偶图（完全二分图）的定义

### Definition

在偶图  $G = \langle V_1, E, V_2 \rangle$  中，若  $V_1$  中的每个结点与  $V_2$  中的每个结点都有且仅有唯一条边相关联，则称偶图  $G$  为完全偶图或完全二分图，记为  $K_{i,j}$ ，其中， $i = |V_1|, j = |V_2|$ 。

### Example



<https://blog.csdn.net/hpdizu80100>

## 偶图的充分必要条件

### Theorem

无向图  $G = \langle V, E \rangle$  为偶图的充分必要条件是所有回路的长度均为偶数。

### Proof.

■ 必要性：令  $C = v_0v_1v_2 \dots v_kv_0$  是偶图  $G = \langle V_1, E, V_2 \rangle$  的任意一条回路，其长度为  $k+1$ 。不妨设  $v_0 \in V_1$ ，由偶图的定义知， $v_1 \in V_2, v_2 \in V_1$ ，依次类推。又因  $v_0 \in V_1$ ，所以  $v_k \in V_2$ ，因而  $k$  为奇数，故  $C$  的长度为偶数。

■ 充分性：设  $G$  中每条回路的长度均为偶数，若  $G$  是连通图（否则可对  $G$  的每个连通分支继续如下论证），任选  $v_0 \in V$ ，定义  $V$  的两个子集如下： $V_1 = \{v_j | d(v_0, v_j) \text{ 为偶数}\}$ ， $V_2 = V - V_1$ 。  
观察  $V_1$  中任两结点间无边存在。假若存在一条边  $(v_i, v_j) \in E$ ，其中  $v_i, v_j \in V_1$ ，则由  $v_0$  到  $v_i$  间的短程线（长度为偶数）以及边  $(v_i, v_j)$ ，再加上  $v_j$  到  $v_0$  间的短程线（长度为偶数）所组成的回路的长度为奇数，与假设矛盾。

同理可证  $V_2$  中任两结点间无边存在。

故  $G$  中每条边  $(v_i, v_j)$ ，必有  $v_i \in V_1, v_j \in V_2$  或  $v_i \in V_2, v_j \in V_1$ ，因此  $G$  是偶图。<https://blog.csdn.net/hpdizu80100>

### 注意

- 根据偶图的充分必要条件，我们可将平凡图和零图看成特殊的偶图。
- 我们常使用它的逆否命题来判断一个图不是偶图：无向图  $G$  不是偶图的充分必要条件是  $G$  中存在长度为奇数的回路。

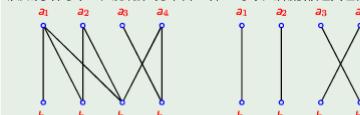


<https://blog.csdn.net/hpdizu80100>

## 匹配的引入

### Example

假设有 4 个工人  $a_1, a_2, a_3, a_4$ ，4 项工作任务  $b_1, b_2, b_3, b_4$ ，并且工人  $a_1$  熟悉任务  $b_1, b_2, b_3$ ； $a_2$  熟悉任务  $b_2, b_3$ ； $a_3$  熟悉任务  $b_4$ ； $a_4$  熟悉任务  $b_3, b_4$ ；建立偶图如下。那么，该如何给每个人分配任务，并且保证每个人做的是自己熟悉的任务呢？



右图就是一种分配方案，称作原图的一个匹配。

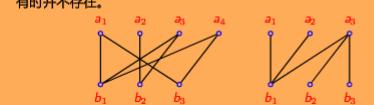
<https://blog.csdn.net/hpdizu80100>

## 偶图的匹配

### Definition

在偶图  $G = \langle V_1, E, V_2 \rangle$  中， $V_1 = \{v_1, v_2, \dots, v_q\}$ ，若存在  $E$  的子集  $E' = \{(v_1, v'_1), (v_2, v'_2), \dots, (v_q, v'_q)\}$ ，其中  $v'_1, v'_2, \dots, v'_q$  是  $V_2$  中的  $q$  个不同的结点，则称  $G$  的子图  $G' = \langle V_1, E', V_2 \rangle$  为从  $V_1$  到  $V_2$  的一个完全匹配，简称匹配。

匹配实际上就是在偶图  $G = \langle V_1, E, V_2 \rangle$  中，寻找  $V_1$  到  $V_2$  的单射。显然，这样的单射有时并不存在。



## 匹配的判定条件 (霍尔定理, t条件)

### Theorem (霍尔定理)

偶图  $G = \langle V_1, E, V_2 \rangle$  中存在从  $V_1$  到  $V_2$  的匹配的充分必要条件是  $V_1$  中任意  $k$  个结点至少与  $V_2$  中的  $k$  个结点相邻,  $k = 1, 2, \dots, |V_1|$ 。这个条件通常称为相异性条件 (diversity condition)。

### Theorem (t 条件)

设  $G = \langle V_1, E, V_2 \rangle$  是一个偶图。如果满足:

- ①  $V_1$  中每个结点至少关联  $t$  条边; ( $V_1$  中结点的最小度数)
- ②  $V_2$  中每个结点至多关联  $t$  条边; ( $V_2$  中结点的最大度数)

则  $G$  中存在从  $V_1$  到  $V_2$  的匹配。其中  $t$  为正整数。这个条件通常称为  $t$  条件 ( $t$ -condition)。

<https://blog.csdn.net/hpdizu80100>

## 匹配的应用

### Example

现有三个课外小组: 物理组, 化学组和生物组, 有五个学生  $s_1, s_2, s_3, s_4, s_5$ :

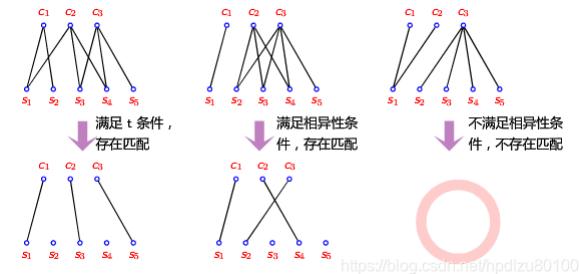
- ①  $s_1, s_2$  为物理组成员,  $s_3, s_4, s_5$  为化学组成员。
- ②  $s_1$  为物理组成员,  $s_2, s_3, s_4$  为化学组成员,  $s_5$  为生物组成员。
- ③  $s_1$  即为物理组成员, 又为化学组成员,  $s_2, s_3, s_4, s_5$  为生物组成员。

在以上三种情况的每一种情况下, 在  $s_1, s_2, s_3, s_4, s_5$  中选三位组长, 不兼职, 问能否办到?

### Solution

用  $c_1, c_2, c_3$  分别表示物理组、化学组和生物组。令  $V_1 = \{c_1, c_2, c_3\}$ ,  $V_2 = \{s_1, s_2, s_3, s_4, s_5\}$ , 以  $V_1, V_2$  为互补结点子集, 以  $E = \{(c_i, s_j) | c_i \in V_1, s_j \in V_2, c_i \text{中有成员 } s_j\}$  为边集, 构造偶图, 然后在这些偶图中寻找匹配。

<https://blog.csdn.net/hpdizu80100>



<https://blog.csdn.net/hpdizu80100>

[理解充分条件、充分不必要条件、必要条件和必要不充分条件 - luciusCheung - 博客园 \(cnblogs.com\)](#)

<https://blog.csdn.net/hpdizu80100/article/details/103254513>

## 欧拉图

### 基本概念

回路: 一条路径的起止顶点相同

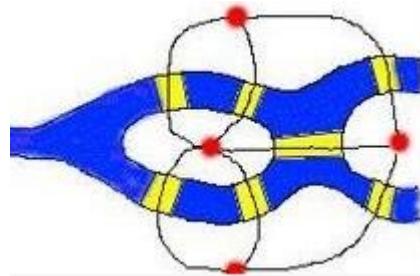
开路: 一条路径的起止顶点不相同

通过图G的每条边一次且仅一次的回路称为欧拉回路。存在欧拉回路的图, 称为欧拉图。

通过图G的每条边一次且仅一次的开路称为欧拉路, 对应的有半欧拉图。

### 起源历史

图论起源于18世纪, 1736年瑞士数学家欧拉 (Euler) 发表了图论的第一篇论文“哥尼斯堡。在当时的哥尼斯堡城有一条横贯全市的普雷格尔河, 河中的两个岛与两岸用七座桥连结起来。当时那里的居民热衷于一个难题: 有游人怎样不重复地走遍七桥, 最后回到出发点。为了解决这个问题, 欧拉用A, B, C, D4个字母代替陆地, 作为4个顶点, 将联结两块陆地的桥用相应的线段表示, 于是哥尼斯堡 七桥问题就变成了图中, 是否存在经过每条边一次且仅一次, 经过所有的顶点的回路问题了。欧拉在论文中指出, 这样的回路是不存在的。



## 相关定理

要想一个图G是**欧拉图**, 图G需要满足两个条件:

- 有向图:
  - 图G是连通的, 不能有孤立的点存在。
  - 每个顶点的入度要等于出度。
- 无向图:
  - 图G是连通的, 不能有孤立的点存在。
  - 度数为奇数的点的个数为0。

要想一个图G是**半欧拉图**, 图G需要满足两个条件:

- 有向图:
  - 图G是连通的, 不能有孤立的点存在。
  - 存在两个顶点, 其入度不等于出度, 其中一点出度比入度大1, 为路径起点, 另一点入度比出度大1, 为路径的终点
- 无向图:
  - 图G是连通的, 不能有孤立的点存在。
  - 度数为奇数的点的个数为2, 并且这两个点一定是路径的起点和终点。

## 具有欧拉通路

欧拉图

(含欧拉回路)

半欧拉图

(不含欧拉回路)

CSDN @Stay-hungry

[欧拉图的基本概念以及判定方法 怎么判断是不是欧拉图-CSDN博客](#)

[离散数学9：二部图、欧拉图、哈密顿图-CSDN博客](#)

## 双连通图

### 概念

是指一个图中任意两个顶点之间均存在至少两条相互独立的路径。换句话说，在双连通图中移除任意一个顶点后，图中仍保持连通（不存在割点/关节点）。

具体来说，一个图G是双连通的，当且仅当满足以下两个条件：

1. G是连通的，即任意两个顶点之间都存在路径相连。
2. 移除G中的任意一个顶点，剩余的子图仍然是连通的。

双连通图有很多重要的性质和应用，例如：

- 在网络系统中，双连通图可以提供高可靠性，因为即使有一个节点失效，网络通信仍然能保持畅通。
- 许多重要的算法如Tarjan算法都依赖于图的双连通性质。
- 双连通图有着很好的视觉特征，在信息可视化中常被用来展示复杂系统的拓扑结构。

## 5、哈希表

### 哈希法概念

哈希法又称散列法、杂凑法、关键字地址计算法。相对应的表称为哈希表、散列表、或杂凑表等。

### 哈希法思想

首先在元素的关键字k和元素的存储位置p之间建立一个对应关系H，使得  $p = H(k)$ ，H成为哈希函数。创建哈希表时，把关键字为k的元素直接存入地址为H(k)的单元，以后查找关键字为k的元素时，再利用哈希函数计算该元素的存储位置。再按关键字存取元素。Hash中存储的key值都是唯一的。

### 哈希冲突

当关键字集合很大的时候，关键字值不同的元素可能会映射到哈希表的同一个地址，即  $k_1 \neq k_2, H(k_1) == H(k_2)$ ，这种现象称为冲突，此时  $k_1$  和  $k_2$  为同义词。事实上冲突是不可避免的，由于关键字可能发生冲突的集合远远大于实际开辟的哈希表长度，构成冲突的必然性，可通过改进哈希的性能来减少冲突，即降低冲突的可能性。

### 优缺点

优点：查找速度极快 ( $O(1)$ )，查找效率与元素个数n无关！

缺点：空间效率低

### 构造原则

- 所选函数尽可能简单，以便提高转换速度；
- 所选函数对关键码计算出的地址，应在哈希地址集中大致均匀分布，以减少空间浪费。

### 数值哈希函数构造、哈希冲突的常用处理方法

#### 哈希函数的构造方法

##### (1) 直接定址法

$Hash(key) = a * key + b$  ( $a, b$  为常数)

优点：以关键码key的某个线性函数值为哈希地址，不会产生冲突。

缺点：要占用连续地址空间，空间效率低。

举例：

关键码集合为{100, 300, 500, 700, 800, 900}，  
选取哈希函数为Hash(key)=key/100，  
则存储结构（哈希表）如下：

0	1	2	3	4	5	6	7	8	9
	100	300		500		700	800	900	

## (2) 除留余数法

Hash(key)=key mod p (p是一个整数)

特点：以关键码除以p的余数作为哈希地址。

关键：如何选取合适的p？

技巧：若设计的哈希表长为m，则一般取p≤m且为质数

## (3) 数字分析法

特点：某关键字的某几位组合成哈希地址。所选的位应当是：各种符号在该位上出现的频率大致相同

举例

有一组（例如80个）关键码，其样式如下：

3	4	7	0	5	2	4
3	4	9	1	4	8	7
3	4	8	2	6	9	6
3	4	8	5	2	7	0
3	4	8	6	3	0	5
3	4	9	8	0	5	8
3	4	7	9	6	7	1
3	4	7	3	9	1	9

位号：① ② ③ ④ ⑤ ⑥ ⑦

讨论：

- ① 第1、2位均是“3和4”，第3位也只能用“7、8、9”，因此，这几位不能用，余下四位分布较均匀，可作为哈希地址选用。  
② 若哈希地址取两位（因元素仅80个），则可取这四位中的任意两位组合成哈希地址，也可以取其中两位与其它两位叠加求和后，取低两位作哈希地址。

## (4) 平方取中法

特点：对关键码平方后，按哈希表大小，取中间的若干位作为哈希地址

理由：因为中间几位与数据的每一位都相关

例：2589的平方值为6702921，可以取中间的029为地址

## (5) 移位法

特点：将关键码自左到右分成位数相等的几部分（最后一部分位数可以短些），然后将这几部分叠加求和，并按哈希表表长，取后几位作为哈希地址。

适用于：每一位上各符号出现概率大致相同的情况

## (6) 折叠法

将各部分的最后一对齐相加间界叠加法，从一端向另一端沿分割界来回折叠后，最后一位对齐相加

举例：元素42751896

移位法：427 + 518 + 96=1041

间界叠加法：427 518 96—> 724+518+69 =1311

## (7) 随机数法

Hash(key)=random(key) (random为随机函数)

适用于：关键字长度不等的情况，造表和查找都很方便

## 解决冲突的方法

### 开放定址法

设计思路：有冲突时就去寻找下一个空的哈希地址，只要哈希表足够大，空的哈希地址总能找到，并将数据元素存入。

#### 1.线性探测法

$$H_i = (\text{Hash}(key) + di) \bmod m \quad (1 \leq i < m)$$

其中： Hash(key)为哈希函数、m为哈希表长度、di为增量序列 1, 2, ..., m-1，且  $di=i$ (含义：一旦冲突，就找附近(下一个)空地址存入)

举例：

关键码集为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，

设：哈希表表长为  $m=11$ ；

哈希函数为  $\text{Hash}(key) = key \bmod 11$ ；

拟用线性探测法处理冲突。建哈希表如下：

0	1	2	3	4	5	6	7	8	9	10
11	22		47	92	16	3	7	29	8	

解释：

① 47、7 (以及11、16、92) 均是由哈希函数得到的没有冲突的哈希地址；

②  $\text{Hash}(29)=7$ ，哈希地址有冲突，需寻找下一个空的哈希地址：由  $H_1 = (\text{Hash}(29)+1) \bmod 11 = 8$ ，哈希地址8为空，因此将29存入。

③ 另外，22、8、3同样在哈希地址上有冲突，也是由  $H_i$  找到空的哈希地址的。

其中3还连续移动了两次 (二次聚集)

线性探测法的优点：只要哈希表未被填满，保证能找到一个空地址单元存放有冲突的元素

线性探测法的缺点：可能使第i个哈希地址的同义词存入第*i+1*个哈希地址，这样本应存入第*i+1*个哈希地址的元素变成了第*i+2*个哈希地址的同义词，……，

因此，可能出现很多元素在相邻的哈希地址上“堆积”起来，大大降低了查找效率

#### 2.二次探测法

$$H_i = (\text{Hash}(key) \pm di) \bmod m$$

其中： Hash(key)为哈希函数、m为哈希表长度，m要求是某个  $4k+3$  的质数；di为增量序列  $1^2, -1^2, 2^2, -2^2, \dots, q^2$

0	1	2	3	4	5	6	7	8	9	10
11	22	3	47	92	16		7	29	8	

注：只有3这个关键码的冲突处理与上例不同，

$\text{Hash}(3)=3$ ，哈希地址上冲突，由

$H_1 = (\text{Hash}(3)+1^2) \bmod 11 = 4$ ，仍然冲突；

$H_2 = (\text{Hash}(3)-1^2) \bmod 11 = 2$ ，找到空的哈希地址，存入。

#### 3.伪随机探测法

若  $di$  为伪随机序列，就称为伪随机探测法

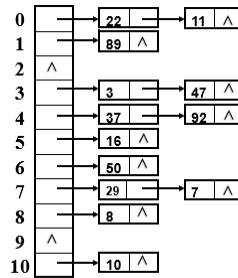
#### 4.链地址法(拉链法)

基本思想：将具有相同哈希地址的记录链成一个单链表，m个哈希地址就设m个单链表，然后用一个数组将m个单链表的表头指针存储起来，形成一个动态的结构。

举例

例：设{47, 7, 29, 11, 16, 92, 22, 8, 3, 50, 37, 89}的哈希函数为：  
 $\text{Hash(key)} = \text{key mod } 11$ ，  
 用拉链法处理冲突，则建表如右图所示。

注：有冲突的元素可以插在表尾，也可以插在表头



## 5. 再哈希法(双哈希函数法)

$$H_i = R H_i(\text{key}) \quad i=1, 2, \dots, k$$

$R H_i$  均是不同的哈希函数，当产生冲突时就计算另一个哈希函数，直到冲突不再发生。

优点：不易产生聚集

缺点：增加了计算时间、建立公共溢出区

思路：除设立哈希基本表外，另设一个溢出向量表，所有关键字和基本表中关键字为同义词的记录，不管它们由哈希函数得到的地址是什么，一旦发生冲突，都填入溢出表。

冲突是不是比较讨厌

不一定

正因为有冲突，使得文件加密后无法破译（不可逆，是单向散列函数，可用于数字签名）。

利用了哈希表性质：源文件稍稍改动，会导致哈希表变动很大。

哈希表的查找效率分析

使用平均查找长度ASL来衡量查找算法啊

ASL取决于

- 哈希函数
- 处理冲突的方法
- 哈希表的装填因子 $\alpha$

$\alpha = \text{表中填入的记录数} / \text{哈希表的长度}$

$\alpha$ 越大，表中记录数越多，说明表装得越满，发生冲突的可能性就越大，查找时比较次数就越多。

ASL与装填因子 $\alpha$ 有关！既不是严格的O(1)，也不是O(n)
$ASL \approx 1 + \frac{\alpha}{2} \quad (\text{拉链法})$
$ASL \approx \frac{1}{2} \left(1 + \frac{1}{1 - \alpha}\right) \quad (\text{线性探测法})$
$ASL \approx -\frac{1}{\alpha} \ln(1 - \alpha) \quad (\text{随机探测法})$

## 结论

- 对哈希表技术具有很好的平均性能，优于一些传统的技术
- 链地址法优于开地址法
- 除留余数法作哈希函数优于其它类型函数

## 字符串哈希函数构造

构造一个数字使之唯一代表一个字符串。但是为了将映射关系进行一一对应，也就是，一个字符串对应一个数字，那么一个数字也对应一个字符串。

用字符串Hash的目的是，我们如果要比较一个字符串，我们不直接比较字符串，而是比较它对应映射的数字，这样子就知道两个“子串”是否相等。从而达到，子串的Hash值的时间为  $O(1)$ ，进而可以利用“空间换时间”来节省时间复杂的。

### 各种字符串Hash函数对比

[字符串哈希 - OI Wiki \(oi-wiki.org\)](#)

[【算法学习】字符串哈希（Hash） 字符串hash-CSDN博客](#)

[哈希函数的常用构造方法 - 楼兰胡杨 - 博客园 \(cnblogs.com\)](#)

[数据结构期末复习-哈希表的6种构造方法和4种解决冲突方法 构造哈希表-CSDN博客](#)

## 2.2.4 算法

### 1、复杂度分析

数据结构和算法效率衡量的标准—**时间复杂度**和**空间复杂度**。

**事前分析估算方法**：在程序运行之前，在程序员编写程序时就进行代码时间复杂度和空间复杂度的估算，进而提高代码运行效率的一种手段。

**事后分析估算方法**：通过设计好的测试程序和数据，来统计和监控，利用计算机计时器对不同算法编制的程序的运行时间进行比较，从而确定算法效率的高低。

**时间复杂度（渐进时间复杂度(asymptotic time complexity)）**：算法的执行时间与数据规模之间的增长关系

**空间复杂度（渐进空间复杂度(asymptotic space complexity)）**：算法的存储空间与数据规模之间的增长关系。

为什么需要复杂度分析，而不是直接使用事后统计法？

1. 事后统计法的测试结果非常依赖测试环境；比如不同的cpu或不同的电脑，运行速度的差异可能非常大，甚至有时候原本在这一台机器上执行A代码的速度比B代码要快，等我们换到另一台机器上时，可能会有截然相反的结果。

2. 测试结果受数据规模影响很大；比如，排序算法，待排序的数据有序度不一样，可能在执行时间上就会有很大的差别。极端情况下，假如数据已经是有序的，那排序算法不需要做任何操作，执行时间就会很短。除此之外，如果测试的数据规模太小，测试结果可能无法真实地反映算法的性能。（比如，对于小规模的数据排序，插入排序可能反倒比快速排序要快）。

所以我们需要一个不用具体的测试数据来测试，就可以粗略地估计算法的执行效率的方法。

### 时间复杂度分析

#### 大O复杂度表示法

假设有所有代码的执行时间  $T(n)$  与每行代码的执行次数成正比，那么大O复杂度表示为： $T(n)=O(f(n))$

$n$  表示数据规模的大小； $f(n)$  表示每行代码执行的次数总和； $O$  代表代码的执行时间  $T(n)$  与  $f(n)$  表达式成正比。

**大O复杂度表示法并不具体表示代码真正的执行时间，而是表示代码执行时间随数据规模增长的变化趋势。所以也叫渐进时间复杂度。**

当n很大时，公式中的低阶、常量、系数三部分并不改变增长的趋势，所以**只需要记录最大的量级就好了**。

1. 只需关注循环执行次数最多的一段代码；
2. 加法法则：总复杂度等于量级最大的那段代码的复杂度；
3. 乘法法则：嵌套代码的复杂度等于嵌套内外代码复杂度的乘积。

## O(1)

一般情况下，只要算法中不存在循环语句，递归语句，即使有成千上万行的代码，其时间复杂度也是O(1)。

## O(n)

算法中有n次循环语句，且其他语句复杂度都为1，这个算法的时间复杂度为O(n)。

## O(logn)、O(nlogn)

```
i=1;
while(i<=n){
    i=i*3;
}
```

诸如此类，时间复杂度为O(logn)，如果此循环外再套一个复杂度为O(n)的循环，则总体的时间复杂度就为O(nlogn)。

## O(m+n)、O(m\*n)

```
int cal(int m, int n) {
    int sum_1 = 0;
    int i = 1;
    for (; i < m; ++i) {
        sum_1 = sum_1 + i;
    }
    int sum_2 = 0;
    int j = 1;
    for (; j < n; ++j) {
        sum_2 = sum_2 + j;
    }

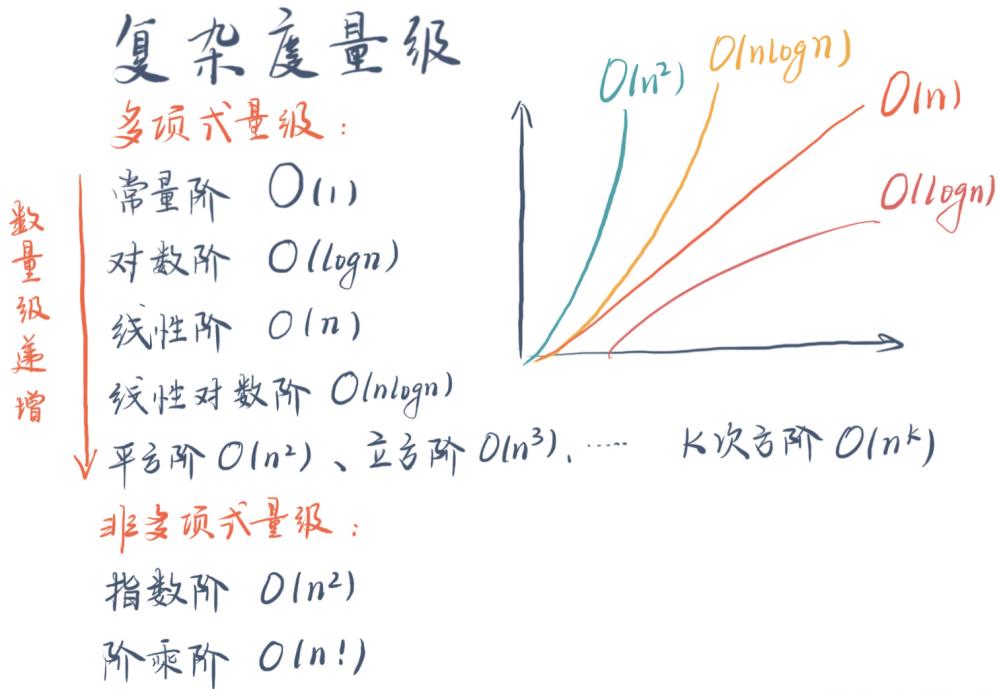
    return sum_1 + sum_2;
}
```

当存在多个数据规模，且无法判断哪个数据规模大的时候，无法直接套用加法法则，需要保留多个数据规模。时间复杂度为O(m+n)。

当循环嵌套时，嵌套代码的复杂度等于嵌套内外代码复杂度的乘积，复杂度为O(m\*n)。

我们常见的空间复杂度就是 O(1) O(n) O(n<sup>2</sup>) 像 O(logn) O(nlogn) 这样的对数阶复杂度平时很少用到。

## 常见时间复杂度与增长速度



## 最好、最坏与平均时间复杂度

**最好情况时间复杂度：**在最理想的情况下，执行这段代码的时间复杂度。

**最坏情况时间复杂度：**在最糟糕的情况下，执行这段代码的时间复杂度。

比如遍历长度为n的数组去找一个特定元素，如果数组第一个元素就是该元素，则时间复杂度为 $O(1)$ ，为最好情况时间复杂度，若最后一个元素是该元素或该元素根本不存在于该数组中，则为最坏情况时间复杂度，为 $O(n)$ 。

依然在长度为n的队列中查找变量x，而我们这一次查找到x之后就直接退出循环。

```
// n表示数组array的长度
int find(int[] array, int n, int x) {
    int i = 0;
    int pos = -1;
    for (; i < n; ++i) {
        if (array[i] == x) {
            pos = i;
            break;
        }
    }
    return pos;
}
```

这种情况下，要查找的变量x，要么在数组里，要么就不在数组里。这两种情况对应的概率统计起来很麻烦，为了方便你理解，我们假设在数组中与不在数组中的概率都为 $1/2$ 。另外，要查找的数据出现在 $0 \sim n-1$ 这n个位置的概率也是一样的，为 $1/n$ 。所以，根据概率乘法法则，要查找的数据出现在 $0 \sim n-1$ 中任意位置的概率就是 $1/(2n)$ 。所以总概率为：

$$\begin{aligned}
 & 1 \times \frac{1}{2n} + 2 \times \frac{1}{2n} + 3 \times \frac{1}{2n} + \dots + n \times \frac{1}{2n} + n \times \frac{1}{2} \\
 = & \frac{3n+1}{4}
 \end{aligned}$$

<https://blog.csdn.net/Charlesssyang>

这个值就是概率论中的加权平均值，也叫作期望值，所以平均时间复杂度的全称应该叫加权平均时间复杂度或者期望时间复杂度。

注意：只有同一块代码在不同的情况下，时间复杂度有量级的差距时！！我们才会使用这三种复杂度表示法来区分。

[复杂度分析——时间复杂度和空间复杂度（超详细总结） 空间复杂度分析-CSDN博客](#)

## 2、算法策略

### 3、基础算法

### 4、排序算法

#### 基数排序

##### 算法思想

基数排序又称为“桶子法”，从低位开始将待排序的数按照这一位的值放到相应的编号为0~9的桶中。等到低位排完得到一个子序列，再将这个序列按照次低位的大小进入相应的桶中，一直排到最高位为止，数组排序完成。

##### 分类

LSD——从低位向高位排

MSD——从高位向低位排

##### 算法执行步骤

- (1)遍历序列找出最大的数(为的是确定最大的数是几位数);
- (2)开辟一个与数组大小相同的临时 数组 `tmp`；
- (3)用一个 `count`数组 统计原数组中某一位(从低位向高位统计)相同的数据出现的次数;
- (4)用一个 `start`数组 计算原数组中某一位(从最低位向最高位计算)相同数据出现的位置;
- (5)将桶中数据从小到大用 `tmp`数组 收集起来;
- (6)重复(3)(4)(5)直到所有位都被统计并计算过，用 `tmp` 收集起来;
- (7)将 `tmp`数组 拷回到原数组中;

##### 图示



## 代码实现

```

int GetMaxDigit(int* arr, size_t n)
{
    assert(arr);
    int digit = 1;
    int base = 10;
    for (size_t i = 0; i < n; i++)
    {
        while (arr[i] >= base)
        {
            ++digit;
            base *= 10;
        }
    }
    return digit;
}

void LSDSort(int* arr, size_t n)
{
    assert(arr);
    int base = 1;
    int digit = GetMaxDigit(arr, n);
    int* tmp = new int[n];
    while (digit--)
    {
        int count[10] = { 0 };
        //统计某一位出现相同数字的个数
        for (size_t i = 0; i < n; i++)
        {
            int index = arr[i] / base % 10;
            count[index]++;
        }
        int start[10] = { 0 };
        //统计个位相同的数在数组arr中出现的位置
        for (size_t i = 1; i < n; i++)
        {
            if (arr[i] / base % 10 == digit)
                start[arr[i] / base % 10]++;
        }
        for (size_t i = 0; i < n; i++)
        {
            arr[i] = tmp[start[arr[i] / base % 10]];
            start[arr[i] / base % 10]++;
        }
    }
    delete[] tmp;
}

```

```

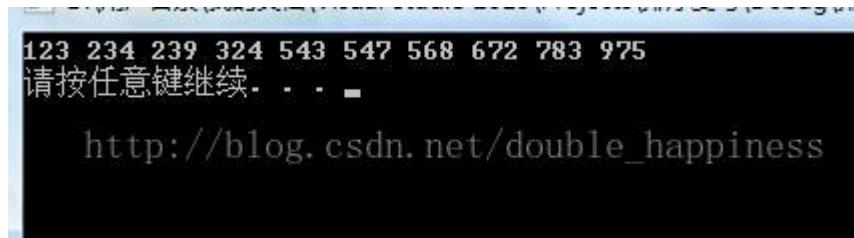
    {
        start[i] = count[i - 1] + start[i - 1];
    }
    //初始化tmp数组
    memset(tmp, 0, n*sizeof(int));
    //从桶中重新排序数据
    for (size_t i = 0; i < n; ++i)
    {
        int index = arr[i] / base % 10;
        tmp[start[index]++] = arr[i];
    }
    //将tmp数组中的元素拷回原数组
    memcpy(arr, tmp, n*sizeof(int));
    base *= 10;
}
delete[] tmp;
}

void Print(int* arr, size_t n)
{
    for (size_t i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}

void TestLSDSort()
{
    int arr[10] = { 123, 234, 543, 324, 568, 975, 547, 672, 783, 239 };
    LSDSort(arr, sizeof(arr) / sizeof(arr[0]));
    Print(arr, sizeof(arr) / sizeof(arr[0]));
}

```

## 运行结果



## 时间复杂度&空间复杂度&稳定性

时间复杂度:  $O(N \cdot digit)$

空间复杂度:  $O(N)$

稳定性: 稳定

## 5、字符串相关算法

### 字符串匹配：KMP 算法

#### 背景

给定一个主串（以 S 代替）和模式串（以 P 代替），要求找出 P 在 S 中出现的位置，此即串的模式匹配问题。

Knuth-Morris-Pratt 算法（简称 **KMP**）是解决这一问题的常用算法之一，这个算法是由高德纳（Donald Ervin Knuth）和沃恩·普拉特在1974年构思，同年詹姆斯·H·莫里斯也独立地设计出该算法，最终三人于1977年联合发表。

#### 真前缀和真后缀

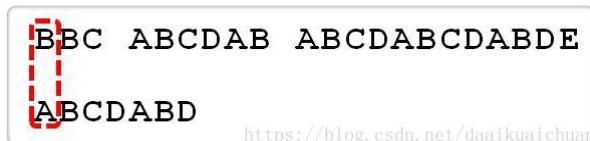


由上图所得，“真前缀”指除了自身以外，一个字符串的全部头部组合；“真后缀”指除了自身以外，一个字符串的全部尾部组合。

#### KMP字符串匹配算法

##### 1、算法流程

(1) 首先，主串“BBC ABCDAB ABCDABCDABDE”的第一个字符与模式串“ABCDABD”的第一个字符，进行比较。因为B与A不匹配，所以模式串后移一位。



(2) 直到主串有一个字符，与模式串的第一个字符相同为止。



(3) 接着比较主串和模式串的下一个字符，直到主串有一个字符，与模式串对应的字符不相同为止。



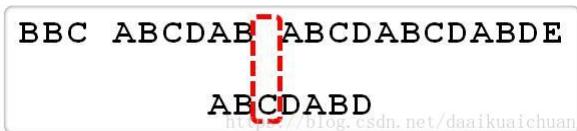
一个基本事实是，当空格与D不匹配时，你其实是已经知道前面六个字符是“ABCDAB”。

KMP算法的想法是，设法利用这个已知信息，不要把“搜索位置”移回已经比较过的位置，而是继续把它向后移，这样就提高了效率。

(4) 怎么做到这一点呢？可以针对模式串，设置一个跳转数组int next[]，这个数组是怎么计算出来的，后面再介绍，这里只要会用就可以了。

i	0	1	2	3	4	5	6	7
模式串	A	B	C	D	A	B	D	'\0'
next[i]	-1	0	0	0	0	1	2	0

(5) 已知空格与D不匹配时，前面六个字符“ABCDAB”是匹配的。根据跳转数组可知，不匹配处D的next值为2，因此接下来从模式串下标为2的位置开始匹配。



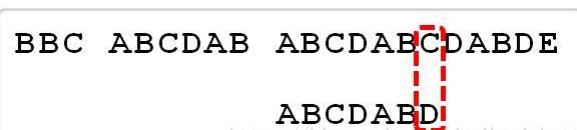
因为空格与C不匹配，**C处的next值为0**，因此接下来模式串从下标为0处开始匹配。

(6)



因为空格与A不匹配，此处**next值为-1**，表示模式串的第一个字符就不匹配，那么直接往后移一位。

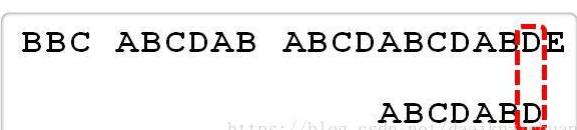
(7)



逐位比较，直到发现C与D不匹配。于是，下一步从下标为2的地方开始匹配。

(8)

逐位比较，直到模式串的最后一一位，发现完全匹配，于是搜索完成。



## 2、next数组是如何求出的

next数组的求解基于“真前缀”和“真后缀”，即next[i]等于P[0]...P[i - 1]最长的相同真前后缀的长度（请暂时忽视i等于0时的情况，下面会有解释）。

如果给定的模式串是：“ABCDABD”，从左至右遍历整个模式串，其各个子串的前缀后缀分别如下表格所示：

模式串的各个子串	前缀	后缀	最大公共元素长度
A	空	空	0
AB	A	B	0
ABC	A,AB	C,BC	0
ABCD	A,AB,ABC	D,CD,BCD	0
ABCDE	A,AB,ABC,ABCD	E,DE,CDE,BDE	1
ABCDAB	A,AB,ABC,ABCD,ABCDAB	B,AB,DAB,CDAB,BCDAB	2
ABCDABD	A,AB,ABC,ABCD,ABCDAB	D,BD,ABD,DABD,CDABD	0

i	0	1	2	3	4	5	6	7
模式串	A	B	C	D	A	B	D	'\0'
next[i]	-1	0	0	0	0	1	2	0

```

1. i = 0 , 对于模式串的首字符 , 我们统一为 next[0] = -1 ;
2. i = 1 , 前面的字符串为 A , 其最长相同真前后缀长度为0 , 即 next[1] = 0 ;
3. i = 2 , 前面的字符串为 AB , 其最长相同真前后缀长度为0 , 即 next[2] = 0 ;
4. i = 3 , 前面的字符串为 ABC , 其最长相同真前后缀长度为0 , 即 next[3] = 0 ;
5. i = 4 , 前面的字符串为 ABCD , 其最长相同真前后缀长度为0 , 即 next[4] = 0 ;
6. i = 5 , 前面的字符串为 ABCDA , 其最长相同真前后缀为 A , 即 next[5] = 1 ;
7. i = 6 , 前面的字符串为 ABCDAB , 其最长相同真前后缀为 AB , 即 next[6] = 2 ;
8. i = 7 , 前面的字符串为 ABCDABD , 其最长相同真前后缀长度为0 , 即 next[7] = 0 ; an

```

那么，为什么根据最长相同真前后缀的长度就可以实现在不匹配情况下的跳转呢？举个代表性的例子：假如 $i = 6$ 时不匹配，此时我们是知道其位置前的字符串为 $ABCDAB$ ，仔细观察这个字符串，首尾都有一个 $AB$ ，既然在 $i = 6$ 处的 $D$ 不匹配，我们为何不直接把 $i = 2$ 处的 $C$ 拿过来继续比较呢，因为都有一个 $AB$ 啊，而这个 $AB$ 就是 $ABCDAB$ 的最长相同真前后缀，其长度2正好是跳转的下标位置。

### 3、next数组的实现

```

void cal_next(string &str, vector<int> &next)
{
    const int len = str.size();
    next[0] = -1;
    int j = -1;
    int i = 0;
    while (i < len - 1)
    {
        if (j == -1 || str[i] == str[j])
        {
            ++j;
            ++i;
            next[i] = j; // 表示第j个字符有k个匹配（“最大长度值” 整体向右移动一位，然后初始值赋为-1）
        }
        else
            j = next[j]; // 往前回溯
    }
}

```

### 三、KMP算法demo

```

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

// 部分匹配表
void cal_next(string &str, vector<int> &next)
{
    const int len = str.size();
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < len - 1)
    {
        if (k == -1 || str[j] == str[k])
        {
            ++k;
            next[j + 1] = k;
        }
        else
            k = next[k];
    }
}

```

```

        ++j;
        next[j] = k; //表示第j个字符有k个匹配（“最大长度值” 整体向右移动一位，然后初始值赋
为-1)
    }
    else
        k = next[k]; //往前回溯
}
}

vector<int> KMP(string &str1, string &str2, vector<int> &next)
{
    vector<int> vec;
    cal_next(str2, next);
    int i = 0; //i是str1的下标
    int j = 0; //j是str2的下标
    int str1_size = str1.size();
    int str2_size = str2.size();
    while (i < str1_size && j < str2_size)
    {
        //如果j = -1, 或者当前字符匹配成功（即s[i] == p[j]）,
        //都令i++, j++. 注意：这里判断顺序不能调换！
        if (j == -1 || str1[i] == str2[j])
        {
            ++i;
            ++j;
        }
        else
            j = next[j]; //当前字符匹配失败，直接从str[j]开始比较，i的位置不变
        if (j == str2_size) //匹配成功
        {
            vec.push_back(i - j); //记录下完全匹配最开始的位置
            j = -1; //重置
        }
    }
    return vec;
}

int main(int argc, char const *argv[])
{
    vector<int> vec(20, 0);
    vector<int> vec_test;
    string str1 = "bacbababababacambabacaddababacasd";
    string str2 = "ababaca";
    vec_test = KMP(str1, str2, vec);
    for (const auto v : vec_test)
        cout << v << endl;
    return 0;
}

```

KMP时间复杂度： $O(m+n)$ 。

#### 四、KMP算法的应用

##### 1、求其中出现重复的任意一个字符

先求next数组， $next[j]=k$ ,  $k > 0$  时，就返回 $j$ ,  $p[j]$ 就是出现重复的字符。

```

while (i < len) {
    if (-1 == j || p[i] == p[j]) {
        ++ i;
        ++ j;
        next[i] = j;
        /* we get it */
        if (j > 0) {
            return i;
        }
    } else {
        j = next[j];
    }
}
https://blog.csdn.net/daaikuaichuan

```

## 2. 求最长的重复子串

求最长的重复子串，就是求 $\text{next}[j]=k$ ，求出 $k$ 的最大值。

```

while (i < len) {
    if (-1 == j || p[i] == p[j]) {
        ++ i;
        ++ j;
        next[i] = j;
        /* we get it */
        if (j > max) {
            max = j;
        }
    } else {
        j = next[j];
    }
}
https://blog.csdn.net/daaikuaichuan

```

<https://segmentfault.com/a/1190000008575379>

<https://blog.csdn.net/willinux20130812/article/details/47133425>

[字符串算法之KMP（字符串匹配） 字符串匹配算法kmp-CSDN博客](#)

## 6、搜索算法

### 搜索的剪枝优化

深度优先搜索（DFS）是一种遍历或搜索树或图的算法，它从一个根节点开始，尽可能深地搜索每个分支，直到找到解为止。在搜索过程中，为了提高效率，减少不必要的搜索，通常会采用各种剪枝优化策略。下面详细介绍几种常用的DFS剪枝优化策略：

#### 1. 优化搜索顺序

- **目的**：通过调整搜索顺序，使得搜索尽快地接近目标或者尽早地剪枝。
- **策略**：在开始搜索前，对所有可能的选择进行排序，优先搜索那些最有可能导致最优解或最快发现无解（以便剪枝）的选项。例如，在解决背包问题时，可以先考虑价值密度最高的物品。

#### 2. 排除等效冗余

- **目的**：避免搜索到重复或等效的解，减少搜索空间。
- **策略**：在搜索过程中，如果发现当前的选择会导致之前已经搜索过的状态，则可以直接剪枝。例如，在全排列问题中，如果某个数已经被选取，那么在这一层的其他分支中就不再考虑这个数。

#### 3. 可行性剪枝

- **目的**：在搜索过程中，一旦发现当前路径不可能达到目标（即不可行），立即停止进一步搜索。
- **策略**：根据问题的约束条件，实时计算当前选择的后果，如果违反约束则立即回溯。例如，在求解和为特定值的子集问题中，如果当前子集的和已经超过目标值，就可以停止向该分支的更深层搜索。

## 4. 最优化剪枝

- **目的**: 在搜索过程中，及早排除那些即使在最好情况下也比已找到的最优解差的路径。
- **策略**: 维护一个全局的最优解变量，在搜索过程中，如果某个分支的最佳可能结果仍然不如当前最优解，则放弃该分支。例如，在旅行商问题（TSP）中，如果当前路径加上未访问城市的最小成本估算仍然大于当前最短路径，则放弃该路径。

### 实践应用

这些剪枝策略在不同的问题中可能需要不同的实现方法和细节调整。剪枝的关键在于如何根据问题的特点和数据特性，合理设计剪枝条件。优秀的剪枝条件能够显著减少搜索空间，提高算法的效率。然而，过度剪枝可能会漏掉正确的解，因此在设计剪枝条件时需要仔细权衡。

在实际编程中，这些剪枝优化策略往往需要根据具体问题进行定制化设计，通过不断试验和调整，找到最适合当前问题的剪枝方法。

<https://www.cnblogs.com/Silly3kidZ/p/18063662>

谈搜索算法的剪枝优化 迷宫问题剪枝优化-CSDN博客

## 记忆化搜索

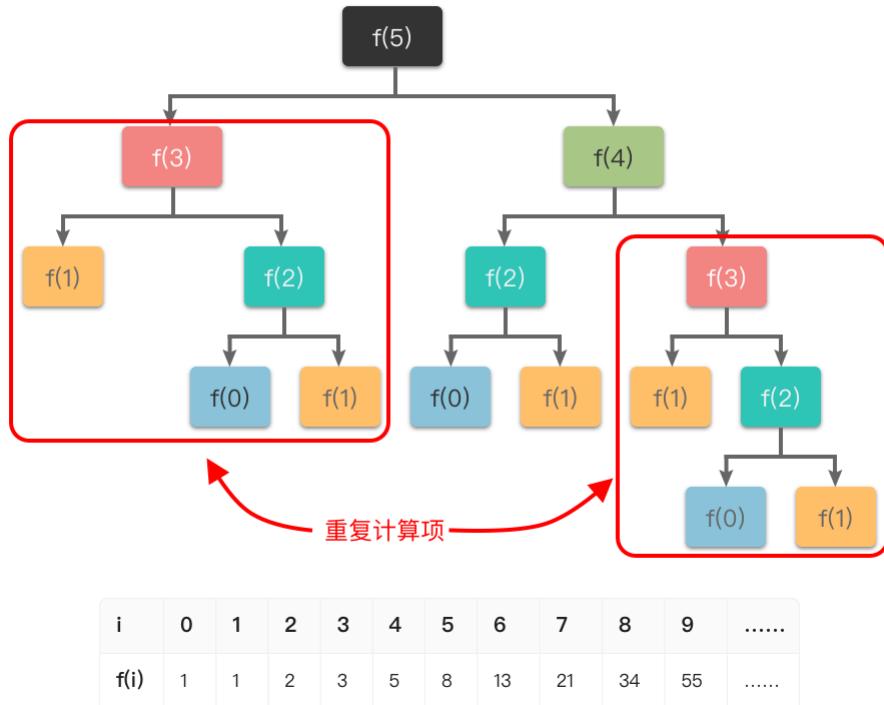
### 概念

**记忆化搜索（Memoization Search）**：是一种通过存储已经遍历过的状态信息，从而避免对同一状态重复遍历的搜索算法。

记忆化搜索是动态规划的一种实现方式。在记忆化搜索中，当算法需要计算某个子问题的结果时，它首先检查是否已经计算过该问题。如果已经计算过，则直接返回已经存储的结果；否则，计算该问题，并将结果存储下来以备将来使用。

举个例子，比如「斐波那契数列」的定义是： $f(0) = 0, f(1) = 1, f(n) = f(n - 1) + f(n - 2)$ 。如果我们使用递归算法求解第  $n$  个斐波那契数，则对应的递推过程如下：

## 记忆化搜索



从图中可以看出：如果使用普通递归算法，想要计算  $f(5)$ ，需要先计算  $f(3)$  和  $f(4)$ ，而在计算  $f(4)$  时还需要计算  $f(3)$ 。这样  $f(3)$  就进行了多次计算，同理  $f(0)、f(1)、f(2)$  都进行了多次计算，从而导致了重复计算问题。

为了避免重复计算，在递归的同时，我们可以使用一个缓存（数组或哈希表）来保存已经求解过的  $f(k)$  的结果。如上图所示，当递归调用用到  $f(k)$  时，先查看一下之前是否已经计算过结果，如果已经计算过，则直接从缓存中取值返回，而不用再递推下去，这样就避免了重复计算问题。

### 记忆化搜索与递推区别

「记忆化搜索」与「递推」都是动态规划的实现方式，但是两者之间有一些区别。

**记忆化搜索：**「自顶向下」的解决问题，采用自然的递归方式编写过程，在过程中会保存每个子问题的解（通常保存在一个数组或哈希表中）来避免重复计算。

- 优点：代码清晰易懂，可以有效的处理一些复杂的状态转移方程。有些状态转移方程是非常复杂的，使用记忆化搜索可以将复杂的状态转移方程拆分成多个子问题，通过递归调用来解决。
- 缺点：可能会因为递归深度过大而导致栈溢出问题。

**递推：**「自底向上」的解决问题，采用循环的方式编写过程，在过程中通过保存每个子问题的解（通常保存在一个数组或哈希表中）来避免重复计算。

- 优点：避免了深度过大问题，不存在栈溢出问题。计算顺序比较明确，易于实现。
- 缺点：无法处理一些复杂的状态转移方程。有些状态转移方程非常复杂，如果使用递推方法来计算，就会导致代码实现变得非常困难。

根据记忆化搜索和递推的优缺点，我们可以在不同场景下使用这两种方法。

适合使用「记忆化搜索」的场景：

1. 问题的状态转移方程比较复杂，递推关系不是很明确。
2. 问题适合转换为递归形式，并且递归深度不会太深。

适合使用「递推」的场景：

1. 问题的状态转移方程比较简单，递归关系比较明确。
2. 问题不太适合转换为递归形式，或者递归深度过大容易导致栈溢出。

## 记忆化搜索解题步骤

1. 写出问题的动态规划「状态」和「状态转移方程」。
2. 定义一个缓存（数组或哈希表），用于保存子问题的解。
3. 定义一个递归函数，用于解决问题。在递归函数中，首先检查缓存中是否已经存在需要计算的结果，如果存在则直接返回结果，否则进行计算，并将结果存储到缓存中，再返回结果。
4. 在主函数中，调用递归函数并返回结果。

## 相关习题

[494. 目标和 - 力扣 \(LeetCode\)](#)

[1137. 第 N 个泰波那契数 - 力扣 \(LeetCode\)](#)

## 启发式搜索

### 概念

启发式搜索(Heuristically Search)又称为有信息搜索(Informed Search)，它是利用问题拥有的启发信息来引导搜索，达到减少搜索范围、降低问题复杂度的目的，这种利用启发信息的搜索过程称为启发式搜索。其代表算法为：贪婪最佳优先搜索(Greedy best-first search)和A\* 搜索。

### A\*搜索

在一个超级大图上动用 Dijkstra 算法，遍历所有的顶点和边，显然会非常耗时。在真实的软件开发中，我们所面对的是超级大的地图和海量的寻路请求，如果算法的执行效率太低，这显然是无法接受的。其实，像出行路线规划、游戏寻路等这些真实软件开发中的问题，在权衡路线规划质量和执行效率的情况下，我们只需要寻求一个次优解就足够了。会用到 A\* 算法。

### 预备知识

#### 状态空间搜索

就是将一个问题的求解过程表现为从初始状态到目标状态寻找一条路径的过程。

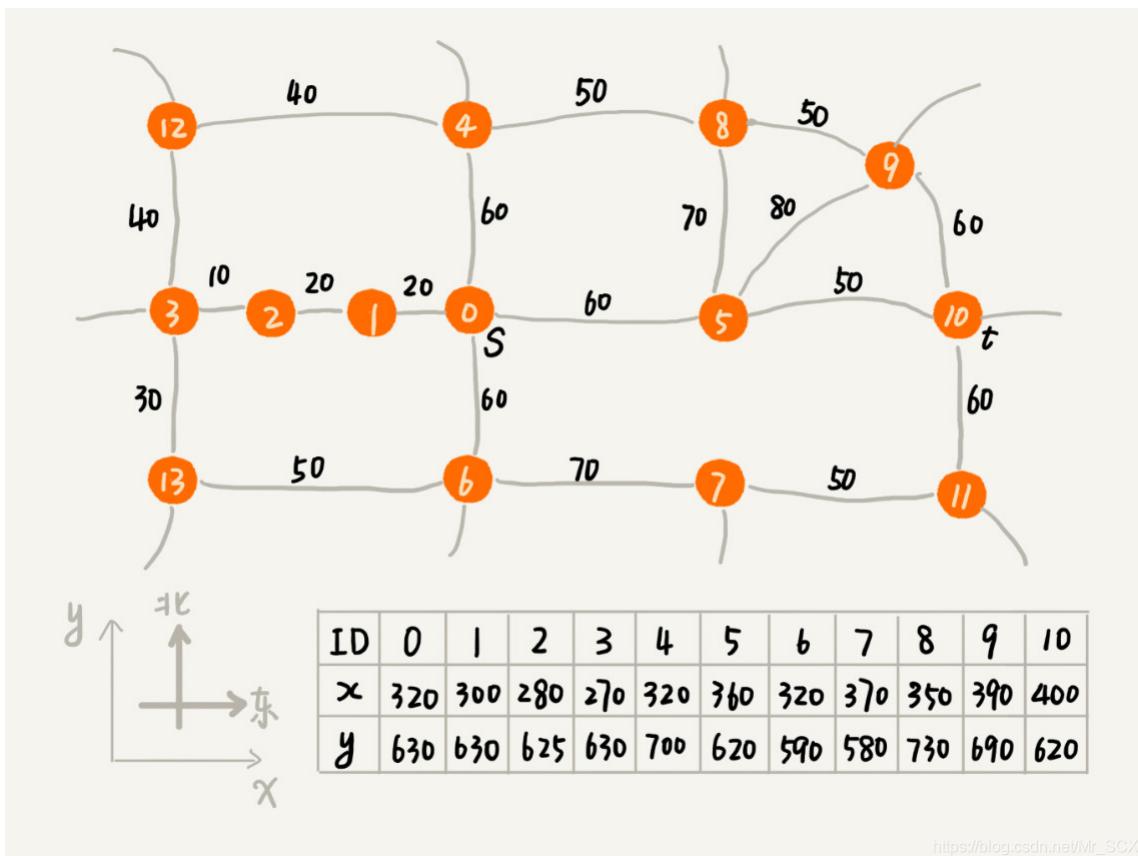
常用的状态空间搜索有广度优先搜索和深度优先搜索，但是它们存在着一个很大的缺陷，就是它们都是在一个给定的状态空间中穷举，都是根据搜索的顺序依次进行搜索，称为盲目搜索，在状态空间不大的时候可以用，但是如果状态空间非常大并且不可预测的情况下就不可取了，它们的效率会非常低。这时就需要用到启发式搜索。

启发式搜索算法，就是在状态空间中先对每一条搜索分支进行评估，得到最好的分支，再从这个分支继续搜索从而到达目标，这样可以有效省略大量无谓的搜索路径，大大提高了搜索效率。在启发式搜索中，对分支的评估是非常重要的，启发式搜索算法定义了一个估价函数  $f(x)$ ，与问题相关的启发式信息都被计算为一定的  $f(x)$  值，引入到搜索过程中。 $f(x) = g(x) + h(x)$ ，其中  $f(x)$  是节点  $x$  的估价函数， $g(x)$  是在状态空间中从初始节点到节点  $x$  的实际代价， $h(x)$  是从节点  $x$  到目标节点的最佳路径的估计代价。 $g(x)$  是已知的，所以在这里主要是  $h(x)$  体现了搜索的启发信息， $h(x)$  专业的叫法是启发函数。换句话说， $g(x)$  代表了搜索的广度的优先趋势。但是当  $h(x) \gg g(x)$  时，可以忽略  $g(x)$ ，从而提高效率。

## 正题

A\* 算法正是利用启发式搜索策略来选择最优的扩展节点，从而提升效率的。实际上，A\* 算法就是在 Dijkstra 算法的基础上进行优化和改造。

回顾一下 Dijkstra 算法的实现思路



在 Dijkstra 算法中，只是单纯地通过顶点  $i$  与起点之间的路径长度  $g(i)$ ，来判断谁先出队列，现在有了这个顶点到终点的路径长度估计值，我们可以通过两者之和  $f(i)=g(i)+h(i)$ ，来判断哪个顶点该最先出队列。综合两部分，我们就能有效避免刚刚讲的“跑偏”。这里  $f(i)$  的专业叫法是估价函数（evaluation function）。

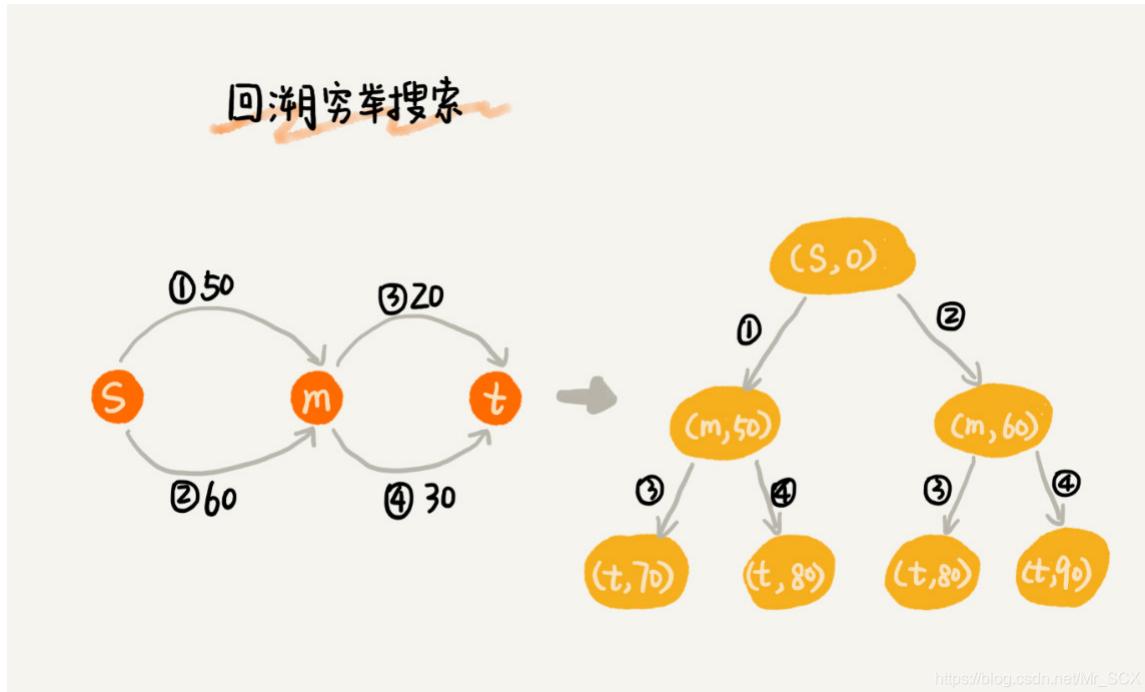
跟 Dijkstra 算法的代码实现，主要有 3 点区别：

- 优先级队列构建的方式不同。A\* 算法是根据  $f(i)$  值 ( $f(i)=g(i)+h(i)$ ) 来构建优先级队列，而 Dijkstra 算法是根据  $dist$  值（也就是刚刚讲到的  $g(i)$ ）来构建优先级队列；
- A\* 算法在更新顶点  $dist$  值的时候，会同步更新  $f(i)$  值；
- 循环结束的条件也不一样。Dijkstra 算法是在终点出队列的时候才结束，A\* 算法是一旦遍历到终点就结束。

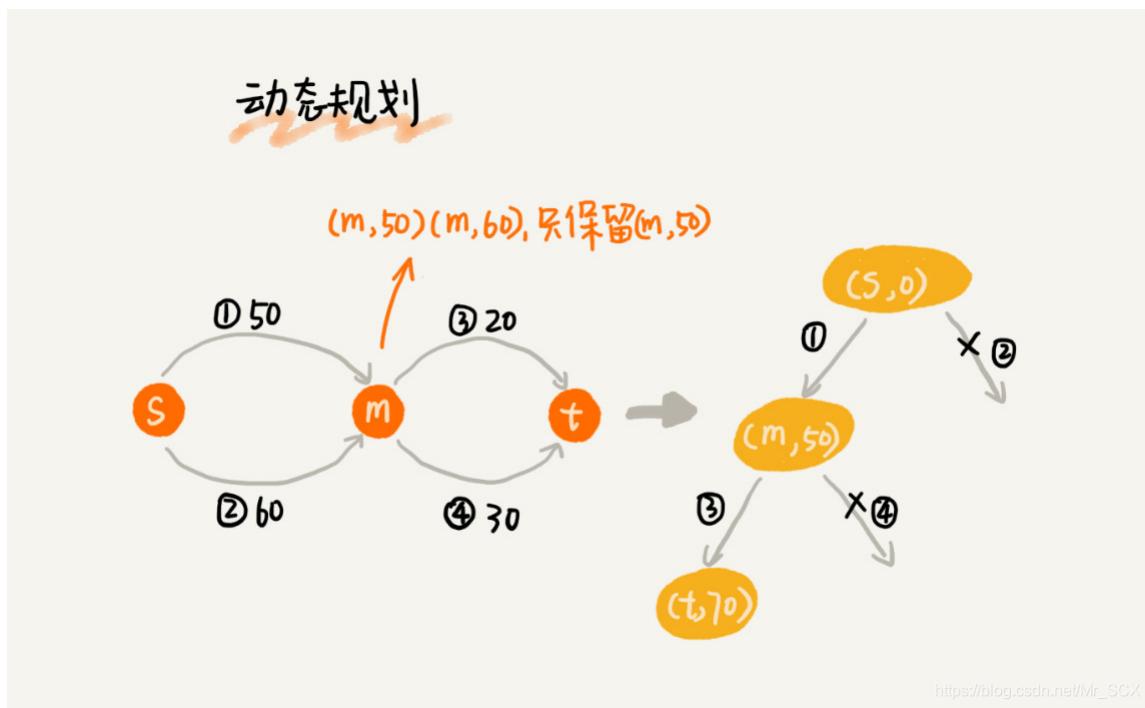
## 思考

尽管 A\* 算法可以更加快速的找到从起点到终点的路线，但是它并不能像 Dijkstra 算法那样，找到最短路线。这是为什么呢？

要找出起点  $s$  到终点  $t$  的最短路径，最简单的方法是，通过回溯穷举所有从  $s$  到达  $t$  的不同路径，然后对比找出最短的那个。不过很显然，回溯算法的执行效率非常低，是指数级的。



Dijkstra 算法在此基础之上，利用动态规划的思想，对回溯搜索进行了剪枝，只保留起点到某个顶点的最短路径，继续往外扩展搜索。动态规划相较于回溯的穷举搜索，只是换了一个实现思路，但它实际上也考察到了所有从起点到终点的路线，所以才能得到最优解。



而 A\* 算法之所以不能像 Dijkstra 算法那样，找到最短路径，主要原因是两者的 while 循环结束条件不一样。刚刚我们讲过，Dijkstra 算法是在终点出队列的时候才结束循环，A\* 算法是一旦遍历到终点就结束循环。对于 Dijkstra 算法来说，当终点出队列的时候，终点的 dist 值是优先级队列中所有顶点的最小值，即便再运行下去，终点的 dist 值也不会再被更新了。对于 A\* 算法来说，一旦遍历到终点，我们就结束 while 循环，这个时候，终点的 dist 值未必是所有顶点中的最小值。换句话说，A\* 算法是利用贪心算法的思路，每次都找 f 值最小的顶点出队列，一旦搜索到终点就清空优先队列，退出while循环，不再继续考察其他顶点和路线了。所以，它并没有考察所有的路线，也就不可能找出最短路径了。

## 小结

### 一、启发式搜索算法

1. 常用的状态空间搜索有广度优先和深度优先搜索算法，但是它们存在着一个很大的缺陷，就是都是在一个给定的状态空间中穷举，都是根据搜索的顺序依次进行搜索，称为盲目搜索，在状态空间不大的时候可以用，但是如果状态空间非常大并且不可预测的情况下就不可取了，它们的效率会非常低。
2. 而启发式搜索算法就是在状态空间中先对每一条搜索分支进行评估，得到最好的分支，再从这个分支继续搜索从而到达目标。在启发式搜索中，利用当前与问题有关的信息作为启发式信息指导搜索，这些信息能够有效省略大量无谓的搜索路径，大大提高了搜索效率。
3. 在启发式搜索中，对分支的评估是非常重要的，如何利用这些启发式信息，启发式搜索算法定义了一个估价函数  $f(x)$ ，与问题相关的启发式信息都被计算为一定的  $f(x)$  的值，引入到搜索过程中。 $f(x) = g(x) + h(x)$ ，其中  $f(x)$  是节点  $x$  的估价函数， $g(x)$  是在状态空间中从初始节点到节点  $x$  的实际代价， $h(x)$  是从节点  $x$  到目标节点的最佳路径的估计代价。 $g(x)$  是已知的，所以在这里主要是  $h(x)$  体现了搜索的启发信息， $h(x)$  专业的叫法是启发函数。换句话说， $g(x)$  代表了搜索的广度的优先趋势。但是当  $h(x) \gg g(x)$  时，可以忽略  $g(x)$ ，从而提高效率。
4. 启发式搜索算法利用估价函数，避免“跑偏”，贪心地朝着最有可能到达终点的方向前进。这种算法找出的路线，并不是最短路线。不过，鉴于启发式搜索算法能很好地平衡路线质量和执行效率，它在实际的软件开发中的应用更加广泛。
5. 算法实例：[A\\* 算法](#)、[IDA\\* 算法](#)、[蚁群算法](#)、[遗传算法](#)、[模拟退火算法](#)等。

### 二、A\* 算法

1. A\* 算法就是对 Dijkstra 算法的简单改造。它跟 Dijkstra 算法的代码实现，主要有 3 点区别：
  - (1) 优先级队列构建的方式不同。A\* 算法是根据估价函数  $f(i)$  的值 ( $f(i)=g(i)+h(i)$ )，其中  $g(i)$  表示顶点  $i$  与起点间的路径长度， $h(i)$  表示顶点  $i$  到终点的路径长度估计值) 来构建优先级队列，而 Dijkstra 算法是根据顶点与起点间的路径长度  $dist$  值 (也就是  $g(i)$ ) 来构建优先级队列的；
  - (2) A\* 算法每次在更新顶点  $i$  的  $dist$  值的时候，也会同步更新  $f(i)$  值；
  - (3) 循环结束的条件也不一样。Dijkstra 算法是在终点出队列的时候才结束，A\* 算法是一旦遍历到终点就结束。
2. 尽管 A\* 算法可以更加快速的找到从起点到终点的路线，但是它并不能像 Dijkstra 算法那样，找到最短路线。A\* 算法实际上是利用贪心算法的思路，每次都找  $f$  值最小的顶点出队列，一旦搜索到终点就不再继续考察其他顶点和路线了。所以，它并没有考察所有的路线，也就不可能找出最短路径了。
3. A\* 算法应用场景：地图 App 中的出行路线规划问题、游戏中人物角色的自动寻路功能。

[https://blog.csdn.net/weixin\\_39059031/article/details/104211475](https://blog.csdn.net/weixin_39059031/article/details/104211475)

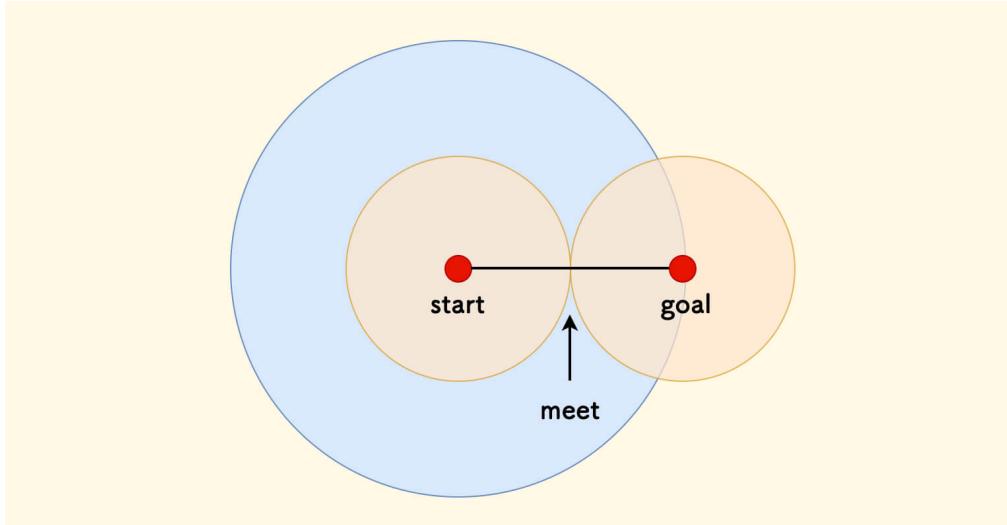
[https://blog.csdn.net/Mr\\_SCX/article/details/104156542](https://blog.csdn.net/Mr_SCX/article/details/104156542)

[A\\*算法详解](#)

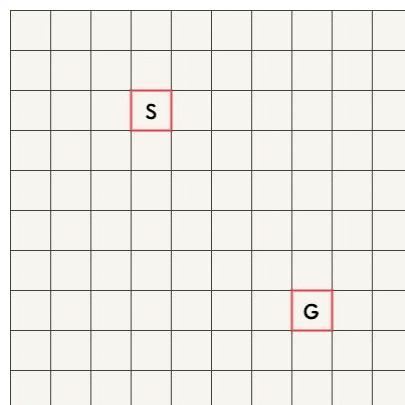
### 双向广度优先搜索

#### 概念

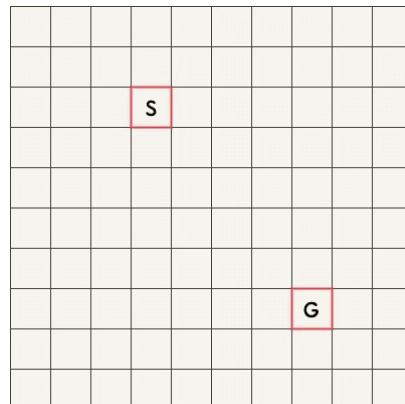
双向广度优先搜索将BFS从起点到终点的单向搜索终点，转换成分别从起点和终点出发的相遇问题。



BFS



双向BFS



如果单向的BFS会超时，那么可以优先考虑双向广搜，以此减少时间复杂度。

### 伪代码

双向 BFS 的 伪代码 形式如下，每次选择 更小的队列 进行扩展：

```

queue<int> q1, q2;
q1.push(start), q2.push(goal);

int steps = 0; // 答案，最少步数

while (!q1.empty() && !q2.empty()) {
    // 每次选择更小的队列进行扩展
}

```

```

    if (q1.size() > q2.size()) swap(q1, q2);
    steps++;
    if (extend(q1)) return steps;
}

bool extend(queue<int>& q) {
    int n = q.size();
    while (n--) {
        auto x = q.front();
        q.pop();
        // TODO: 判断是否和对面相遇, 相遇则返回 true
        // TODO: 向外扩展一层, 加入队列
    }
    return false;
}

```

<https://writings.sh/post/bidirectional-bfs>

[广度优先搜索之双向bfs（实操篇）-CSDN博客](#)

## 迭代加深搜索

### 概念

迭代加深搜索(Iterative Deepening Depth-First Search, IDDFS)

### 伪代码

```

function IDDFS(start, goal):
    for depth = 0 to infinity:
        if DLS(start, goal, depth) == found:
            return found
    return not_found

function DLS(node, goal, limit):
    if node == goal:
        return found
    if limit == 0:
        return cutoff
    cutoff_occurred = false
    for each child of node:
        result = DLS(child, goal, limit-1)
        if result == cutoff:
            cutoff_occurred = true
        else if result == found:
            return found
    if cutoff_occurred:
        return cutoff
    else:
        return not_found

```

1. IDDFS函数从深度0开始不断增加深度限制，调用DLS函数进行深度优先搜索。
2. DLS函数进行深度优先搜索，当达到深度限制时返回“cutoff”标志，表示需要进一步增加深度限制。
3. 如果找到目标节点，返回“found”标志；如果所有子节点都没找到目标节点，返回“not\_found”标志。
4. IDDFS函数不断调用DLS函数，直到找到目标节点或者搜索完所有可能的深度。

这种迭代加深的方法可以充分利用深度优先搜索的空间效率,同时也能渐进地扩大搜索范围,避免陷入过深的搜索路径。这种算法常用于解决棋类游戏、路径规划等问题。

### 迭代加深搜索的优点和局限性

#### 优点

对内存的需求相对较小。这是因为它只需要存储当前的搜索路径,而不需要存储所有的搜索节点。这使得它在处理大规模问题时具有很大的优势。此外,迭代加深搜索能够保证找到的解是最优解,这是因为它会遍历所有的搜索路径,直到找到解为止。

#### 局限性

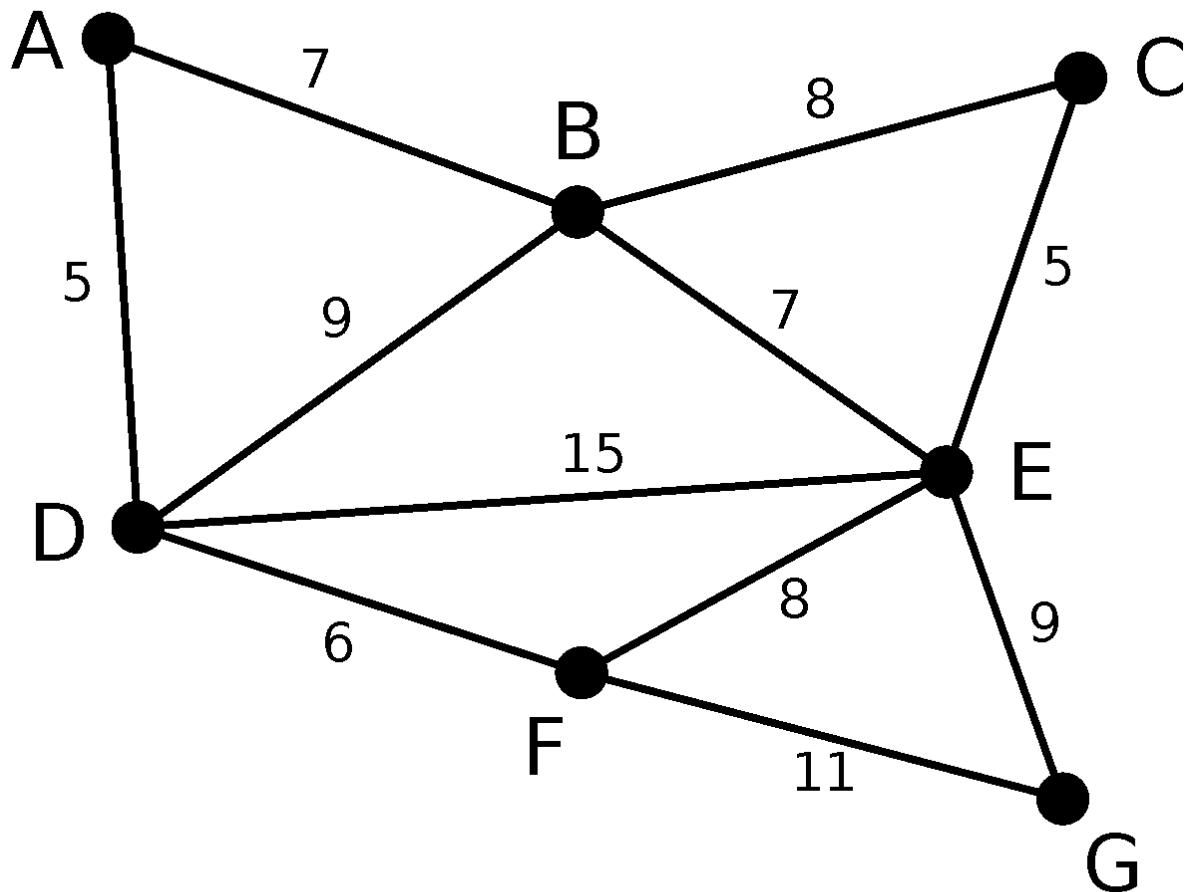
首先,它的运行时间可能会比较长,特别是在搜索空间非常大的情况下。其次,它可能会重复访问一些节点,这会增加搜索的复杂性。最后,它可能无法找到最优解,特别是在搜索空间中存在多个解的情况下。

## 7、图论算法

### 次小生成树

#### 回顾最小生成树算法——Kruskal 算法

Kruskal 算法是一种常见并且好写的最小生成树算法,由 Kruskal 发明。该算法的基本思想是从小到大加入边,是个贪心算法。



其他最小生成树算法: Prim算法、[Boruvka 算法](#)

## 非严格次小生成树

### 定义

在无向图中，边权和最小的满足边权和 **大于等于** 最小生成树边权和的生成树

如果最小生成树选择的边集是  $E_M$ ，严格次小生成树选择的边集是  $E_S$ ，那么需要满足：  
( $value(e)$  表示边  $e$  的权值)

$$\sum_{e \in E_M} value(e) \leq \sum_{e \in E_S} value(e)$$

### 求解方法

- 求出无向图的最小生成树  $T$ ，设其权值和为  $M$
- 遍历每条未被选中的边  $e = (u, v, w)$ ，找到  $T$  中  $u$  到  $v$  路径上边权最大的一条边  $e' = (s, t, w')$ ，则在  $T$  中以  $e$  替换  $e'$ ，可得一棵权值和为  $M' = M + w - w'$  的生成树  $T'$
- 对所有替换得到的答案  $M'$  取最小值即可

如何求  $u, v$  路径上的边权最大值呢？

我们可以使用倍增来维护，预处理出每个节点的  $2^i$  级祖先及到达其  $2^i$  级祖先路径上最大的边权，这样在倍增求 LCA 的过程中可以直接求得。

## 严格次小生成树

### 定义

在无向图中，边权和最小的满足边权和 **严格大于** 最小生成树边权和的生成树

$$\sum_{e \in E_M} value(e) < \sum_{e \in E_S} value(e)$$

### 求解方法

考虑刚才的非严格次小生成树求解过程，为什么求得的解是非严格的？

因为最小生成树保证生成树中  $u$  到  $v$  路径上的边权最大值一定 **不大于** 其他从  $u$  到  $v$  路径的边权最大值。换言之，当我们用于替换的边的权值与原生成树中被替换边的权值相等时，得到的次小生成树是非严格的。

解决的办法很自然：我们维护到  $2^i$  级祖先路径上的最大边权的同时维护 **严格次大边权**，当用于替换的边的权值与原生成树中路径最大边权相等时，我们用严格次大值来替换即可。

这个过程可以用倍增求解，复杂度  $O(m \log m)$ 。

<https://oi-wiki.org/graph/mst/>

## 单源次短路

### 单源最短路径

Bellman-Ford 算法、SPFA 算法、Dijkstra 算法

### 非严格次短路

#### 特点

- 次短路肯定至少有一条边不在最短路上
- 次短路路径长度  $\geq$  最短路路径长度

#### 思路

跑最短路  $\rightarrow$  记录路径  $\rightarrow$  枚举删边，再跑最短路  $\rightarrow$  处理答案

## 严格次短路

特点

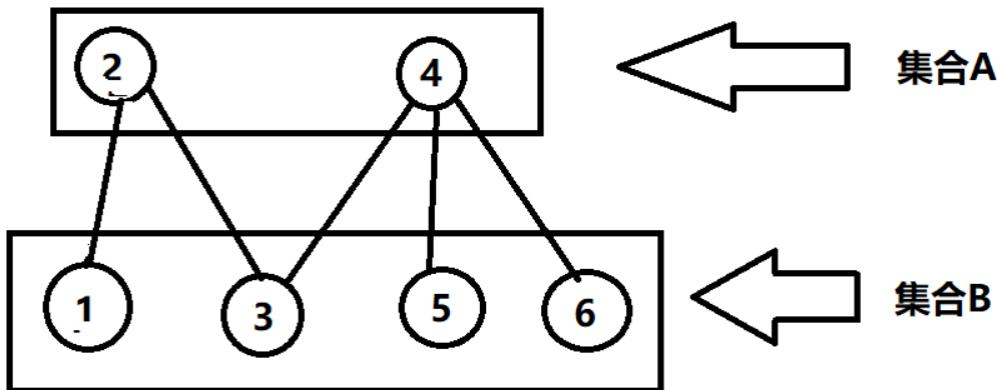
- 次短路肯定至少有一条边不在最短路上
- 次短路路径长度>最短路路径长度

思路

跑最短路 -> 记录路径 -> 枚举删边，再跑最短路 -> 比较路径长度 -> 答案

[次短路与 k 短路 - Zvelig1205 - 博客园 \(cnblogs.com\)](#)

## 二分图的判定



## 染色法+DFS

对于一个无向图，起初图中所有顶点都是无色，我们从任意未染色顶点出发，对这个顶点进行染色，对于与这个顶点相邻的顶点，有三种情况：

- 未染色
  - 那么将这个顶点染色，染与当前顶点不相同的颜色。
- 已染色
  - 与当前顶点颜色不同
    - 跳过
  - 与当前顶点颜色相同
    - 不是二分图

直到图中所有顶点已经进行染色，并且没有出现相邻顶点颜色相同的情况，则该图是一个二分图。

## 并查集

使用并查集来将图中的节点分成两个集合。如果发现相邻节点属于同一个集合，则说明不是二分图。

## 例子

一个无向图  $G = (V, E)$ ，其中  $V = \{1, 2, 3, 4, 5, 6\}$  和  $E = \{(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (4, 5), (4, 6), (5, 6)\}$ 。

割点、割边

树的重心、直径、DFS序与欧拉序

树上差分、子树和与倍增

## 8、动态规划

状态压缩动态规划

动态规划的常用优化

### 2.2.5 数学与其他

#### 1、初等数学

代数（高中部分）

几何（高中部分）

#### 2、初等数论

同余式

欧拉定理和欧拉函数

费马小定理

威尔逊定理

裴蜀定理

模运算意义下的逆元

扩展欧几里得算法

中国剩余定理

#### 3、离散与组合数学

多重集合

等价类

多重集上的排列

多重集上的组合

错排列、圆排列

鸽巢原理

二项式定理

容斥原理

卡特兰 (Catalan) 数

## 4、线性代数

向量与矩阵的概念

向量的运算

矩阵的初等变换

矩阵的运算：加法、减法、乘法与转置

特殊矩阵的概念：单位阵、三角阵、对称阵和稀疏矩阵

高斯消元法

### 2.3 NOI 级

---