

目 录

致谢

Introduction

Commitors

Preface 前言

Chapter-1 Sort 第1章 排序

InsertSort 插入排序

BubbleSort 冒泡排序

QuickSort 快速排序

MergeSort 归并排序

Chapter-2 Search 第2章 搜索

KnowledgePoint 知识要点

BinarySearch 二分查找法 (折半查找法)

BruteForce 暴力枚举

Recursion 递归

BreadthFirstSearch 广度优先搜索

BidirectionalBreadthSearch 双向广度搜索

AStarSearch A*搜索

DancingLinks 舞蹈链

Chapter-3 DataStructure 第3章 数据结构

DisjointSet 并查集

FenwickTree(BinaryIndexTree) 树状数组

SegmentTree 线段树

LeftistTree(LeftistHeap) 左偏树 (左偏堆)

PrefixTree 前缀树

SuffixTree 后缀树

AVLTree AVL平衡树

RedBlackTree 红黑树

Chapter-4 DynamicProgramming 第4章 动态规划

KnowledgePoint 知识要点

Section-2 KnapsackDP 第2节 背包问题

ZeroOneKnapsack 01背包

ZeroOneKnapsackExtension 01背包扩展

CompleteKnapsack 完全背包

[TwoDimensionKnapsack 二维背包](#)

[GroupKnapsack 分组背包](#)

Section-3 RegionalDP 第3节 区域动规

[MinimumMergeCost - 最小合并代价](#)

[MinimumMergeCostExtension - 最小合并代价扩展](#)

[MaximumBinaryTreeMerge - 最大二叉树合并](#)

Section-4 TreeDP 第4节 树形动规

[BinaryTreeDP 二叉树动规](#)

[MultipleTreeDP 多叉树动规](#)

[MultipleTreeDPExtension 多叉树动规扩展](#)

[LoopedMultipleTreeDP 带环多叉树动规](#)

[TraverseBinaryTreeDP 遍历二叉树动规](#)

Chapter-5 GraphTheory 第5章 图论

Section-1 Traverse 第1节 遍历

[KnowledgePoint 知识要点](#)

[PreorderTraverse 先序遍历](#)

[InorderTraverse 中序遍历](#)

[PostorderTraverse 后序遍历](#)

[LevelorderTraverse 层序遍历](#)

[DepthFirstSearch 深度优先搜索](#)

[BreadthFirstSearch 广度优先搜索](#)

[TopologicalSort 拓扑排序](#)

[EulerCycle 欧拉回路](#)

Section-2 MinimumSpanningTree 第2节 最小生成树

[KnowledgePoint 知识要点](#)

[Kruskal Kruskal算法](#)

[Prim Prim算法](#)

[SecondMinimumSpanningTree 次小生成树](#)

[OptimalRatioSpanningTree 最优比率生成树](#)

Section-3 ShortestPath 第3节 最短路径

[KnowledgePoint 知识要点](#)

[Relaxation 松弛操作](#)

[BellmanFord BellmanFord算法](#)

[ShortestPathFasterAlgorithm 最短路径更快算法 \(SPFA\)](#)

[Dijkstra Dijkstra算法](#)

[Floyd Floyd算法](#)

[DifferentConstraints 差分约束](#)

Section-4 Connectivity 第4节 连通

[KnowledgePoint 知识要点](#)

[Kosaraju Kosaraju算法](#)

[Tarjan Tarjan算法](#)

[Gabow - Gabow算法](#)

[TwoSatisfiability 2-SAT问题](#)

[Cut 割](#)

[双向递增递减子序列](#)

[DoubleConnectedComponent 双联通分支](#)

[LeastCommonAncestor 最近公共祖先](#)

[RangeExtremumQuery 区域最值查询](#)

Section-5 FlowNetwork 第5节 网络流

[EdmondsKarp EdmondsKarp算法](#)

[PushAndRelabel 压入与重标记](#)

[Dinic Dinic算法](#)

[DistanceLabel 距离标号算法](#)

[RelabelToFront 重标记与前移算法](#)

[HighestLabelPreflowPush 最高标号预留与推进算法](#)

[DistanceLabel-AdjacentListVersion 距离标号算法-邻接表优化版](#)

[Summary-Maxflow 最大流算法小结](#)

[MinimumCost-Maxflow 最小费用最大流](#)

[MultipleSourceMultipleSink-Maxflow 多源点、多汇点的最大流](#)

[Connectivity 连通度](#)

[NoSourceNoSink-VolumeBoundedFlow 无源点、无汇点、容量有上下界的流网络](#)

[VolumeBounded-Maxflow 容量有上下界的最大流](#)

[VolumeBounded-Minflow 容量有上下界的最小流](#)

Section-6 BinaryMatch 第6节 二分匹配

[Hungarian 匈牙利算法](#)

[HopcroftKarp Hopcroft-Karp算法](#)

[MatchToMaxflow 二分匹配转化为最大流](#)

[KuhnMunkres Kuhn-Munkres算法](#)

Introduction-Domination,Independent,Covering,Clique 介绍支配集、独立集、覆盖集和团

WeightedCoveringAndIndependentSet 最小点权覆盖和最大点权独立集

MinimumDisjointPathCovering 最小不相交路径覆盖

MinimumJointPathCovering 最小可相交路径覆盖

Coloring 染色问题

Chapter-6 Calculation 第6章 计算

LargeNumber 大数字

DecimalConversion 进制转换

Exponentiation 幂运算

Chapter-7 CombinatorialMathematics 第7章 组合数学

KnowledgePoint 知识要点

FullPermutation 全排列

UniqueFullPermutation 唯一的全排列

Combination 组合

DuplicableCombination (元素)可重复的组合

Subset 子集

UniqueSubset 唯一的子集

Permutation 排列

PermutationGroup 置换群

Catalan 卡特兰数

Chapter-8 NumberTheory 第8章 数论

Sieve 筛选算法

Euclid 欧几里得

EuclidExtension 欧几里得扩展

ModularLinearEquation 模线性方程

ChineseRemainerTheorem 中国剩余定理

ModularExponentiation 模幂运算

Chapter-9 LinearAlgebra 第9章 线性代数

Section-1 Matrix 第1节 矩阵

Strassen Strassen算法

GaussElimination 高斯消元法

LUP LUP分解

InverseMatrix 矩阵求逆

Section-2 LinearProgramming 第2节 线性规划

Simplex 单纯形算法

Dinkelback Dinkelback算法

Chapter-10 AnalyticGeometry 第10章 解析几何

Section-1 Polygon 第1节 多边形

Cross 叉积

SegmentIntersection 线段相交

PointInConvexPolygon 点在多边形内

Sweeping 扫描算法

ConvexPolygonArea 凸多边形面积

ConvexPolygonGravityCenter 凸多边形重心

RayDistinguish 射线判别

RotatingCalipers 旋转卡壳

Section-2 ConvexHull 第2节 凸包

NearestNeighbor 最近点对

GrahamScan Graham扫描算法

QuickConvexHull 快速凸包算法

Chapter-11 TextMatch 第11章 文本匹配

SimpleMatch 简单匹配

KMPMatch KMP匹配算法

ACAutomation AC自动机

Chapter-12 GameTheory 第12章 博弈论

BashGame 巴什博弈

WythoffGame 威佐夫博弈

NimGame 尼姆博弈

SpragueGrundy SG函数

致谢

当前文档《Way to Algorithm - 算法之路》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建, 生成于 2018-03-12。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能, 以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理, 书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候, 发现文档内容有不恰当的地方, 请向我们反馈, 让我们共同携手, 将知识准确、高效且有效地传递给每一个人。

同时, 如果您在日常生活、工作和学习中遇到有价值有营养的知识文档, 欢迎分享到 书栈(BookStack.CN), 为知识的传承献上您的一份力量!

如果当前文档生成时间太久, 请到 书栈(BookStack.CN) 获取最新的文档, 以跟上知识更新换代的步伐。

文档地址: <http://www.bookstack.cn/books/Way-to-Algorithm>

书栈官网: <http://www.bookstack.cn>

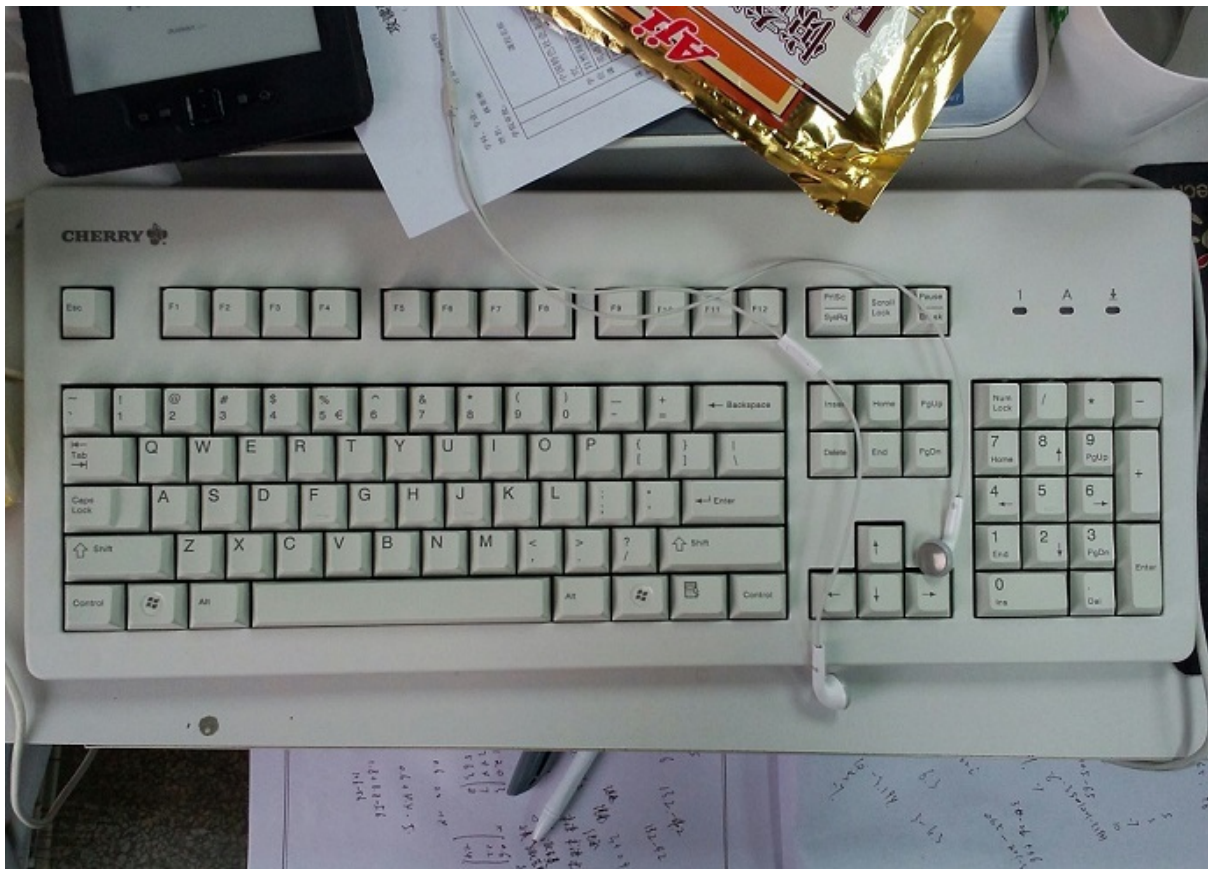
书栈开源: <https://github.com/TruthHun>

分享, 让知识传承更久远! 感谢知识的创造者, 感谢知识的分享者, 也感谢每一位阅读到此处的读者, 因为我们都将成为知识的传承者。

Introduction

- [Introduction](#) - 简介
- [Reference](#) - 参考
- [Contributing](#) - 贡献

Way to Algorithm - 算法之路



Algorithm Tutorial and Source Code - 算法教程与源码

Introduction - 简介

本书围绕大学生计算机算法，收集和整理了比较常见的算法与数据结构写成。借鉴了一些经典的算法书籍，和互联网上的资料。本书通过数学公式和图画的方式来描述算法，并配备源码实现、测试用例，以及一些在Online Judge网站上的题目，来帮助读者真正理解算法。

阅读地址：[Way to Algorithm](#)

由于个人经验有限，难免存在一些疏漏和错误的地方，还请指正。

Reference - 参考

- [算法导论 - Introduction to Algorithms](#)
- [背包问题九讲](#)
- [USACO](#)
- [StackOverflow](#)
- [Wikipedia](#)
- [LeetCode](#)

Contributing - 贡献

见 [Contributing](#)

Commitors

- [Commitors](#)

本书目前只能算是半成品，因为参加工作，项目进度一直比较慢。如果您愿意参与，可以直接fork项目，然后提交pull request，我会review之后merge。

Commitors

- [linrongbin](#)
- [NEWPLAN](#)

Preface 前言

- Preface - 前言

Preface - 前言

在我大学期间刷算法题目的时候，经常遇到的情况是：要么题目太简单一看就会做，要么题目太难想三天也做不出来。

后来我明白了其原因在于我并没有做到真正的深刻理解这些题目背后的算法，只会把一些算法原理套用在最简单的算法题目上面，题目稍微改变，我就没办法搞定它们了。

当时我一边刷OJ题目，一边把自己遇到的各种算法源码进行整理并归类，放在github库上。可过了半年之后再看这些代码，发现我已经完全不认识它们了。

因为如果不理解算法背后的数学意义，只看代码确实很难搞清楚那些代码到底在干什么。

我认为单纯的算法源码很难学习理解，而将算法图形化、公式化才是最容易让人理解的方式，因为可以从数学角度来准确的描述问题和解决过程。

因此我将这些内容用自己的方式画出来，配合代码和测试，就形成了目前这本关于大学生计算机算法的书籍。它不再是简单的“源码”合集，而更加是一本“书”，

本书的每一章专门讲解一类算法问题，其中又划分多个小节，专门讲解其中的一个分支或变种。同一章节中各个小节间会有明显的联系，基础的算法在前面，变种的、高级的算法在后面。每个算法都有讲解、源码和测试三个部分。

在编写的过程中，我学习和参考了非常多的资料，有很多都忘记了，记得的会列在参考资料中。

本书的发布方式也几经周折。我需要一种能够同时展示文字段落、数学公式、插图和源代码这四个部分的载体，来展示自己的想法。之前尝试过用doc编写，到处pdf展示（github上可以直接浏览pdf文档）。但因为二进制文件（pdf）没办法被版本控制软件git去重，导致项目库臃肿，每次更新docx文档，都会生成新的pdf文件。

后来尝试了markdown编写，再用visio绘制png图片，用github page来发布静态文档，这样可以避免生成pdf文件，但png图片仍然无法会占用大量的仓库存储。后来发现SVG这种基于文本文件且支持浏览器的图片可以完美解决，最终采用了draw.io来绘制SVG图片的方式。

最后一个问题就是如何在markdown文档中显示数学公式，当我发现gitbook的插件“mathjax”可以支持Latex，而“include-codeblock”甚至可以在文档中直接展示算法源码时，立刻决定采用gitbook作为本书的发布方式。

gitbook的发布方式支持markdown文档、Latex公式、SVG图片、展示源码、集成github代码库，而且通过gitbook站点可以直接分发本书。但由于gitbook的插件mathjax对website和ebook的支持存在bug，目前通过浏览器访问需要两次刷新才能正确显示Latex公式，而且pdf也无法正确下载。这些bug可能需要更久的时间等待gitbook及相关作者的修复，或者找到其它更好的替换方案。

本书主要使用[markdown](#)编写文档、[draw.io](#)制作SVG插图、[cmake](#)管理C++11项目、[gitbook](#)发布书籍，编写时尽量与书中基本完成的前四章风格保持一致。

后来NEWPLAN同学参加到这本书籍的编写中，他添加了很多内容。

欢迎更多的同学来一起丰富这些资料。也希望可以增加更多专业领域中的计算机算法，不再局限于OJ和算法比赛。

西安交通大学计算机系

林荣彬

2014年2月16日

Chapter-1 Sort 第1章 排序

第1章 排序

1. [InsertSort](#) 插入排序
2. [BubbleSort](#) 冒泡排序
3. [QuickSort](#) 快速排序
4. [MergeSort](#) 归并排序

InsertSort 插入排序

- [Insert Sort - 插入排序](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Insert Sort - 插入排序

问题

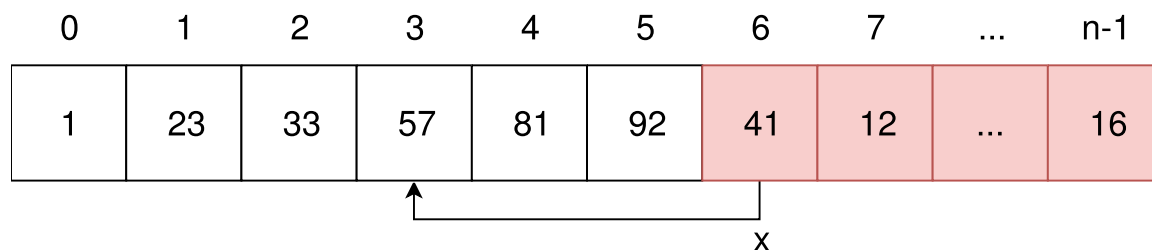
用插入排序对长度为 n 的无序序列 s 进行排序。

解法

本问题对无序序列 s 进行升序排序，排序后 s 是从小到大的。

将长度为 n 的序列 s 分为左右两个部分 $left$ 和 $right$ ， $left$ 是已序的，范围为 $s[0, k]$ ， $right$ 是未序的，范围为 $s[k+1, n-1]$ ，其中 $0 \leq k < n$ 。对 $right$ 中最左边的元素 $x = s[k+1]$ ，在 $left$ 部分中找到一个位置 i ，满足 $s[i-1] \leq x \leq s[i]$ ，也就是说 x 可以夹在 $s[i-1]$ 和 $s[i]$ 之间。为了满足 $left$ 的有序性，将 $left$ 中 $s[i, k]$ 部分的元素向右移动一个位置到 $s[i+1, k+1]$ ，将 x 放置在原 $s[i]$ 的位置即可。

例如下图中， $left$ 部分为 $s[0, 5]$ ， $right$ 部分为 $s[6, n-1]$ ， $right$ 最左边的首部元素 $x = s[6] = 41$ ，在 $left$ 部分中合适的插入位置为 $i = 3$ （ $s[2] \leq x \leq s[3]$ ）。



将 $s[3, 5]$ 向右移动一位到 $s[4, 6]$ ，将原 x 移动到 $s[3]$ ，就完成了插入。

0	1	2	3	4	5	6	7	...	n-1
1	23	33	41	57	81	92	12	...	16

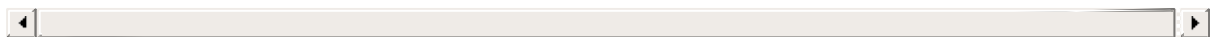
对于长度为 n 的数组 s ，将 $left$ 初始化为 $s[0,0]$ ， $right$ 初始化为 $s[1,n-1]$ 。重复上面的插入操作，直到 $right$ 为空，这时 $left$ 部分即为已序的结果，算法结束。对长度为 n 的序列 s ，每一轮将 $right$ 中一个元素插入 $left$ 中的时间为 $O(n)$ ，总共需要 n 轮操作，该算法的时间复杂度为 $O(n^2)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



BubbleSort 冒泡排序

- [Bubble Sort - 冒泡排序](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Bubble Sort - 冒泡排序

问题

用冒泡排序对长度为 n 的无序序列 s 进行排序。

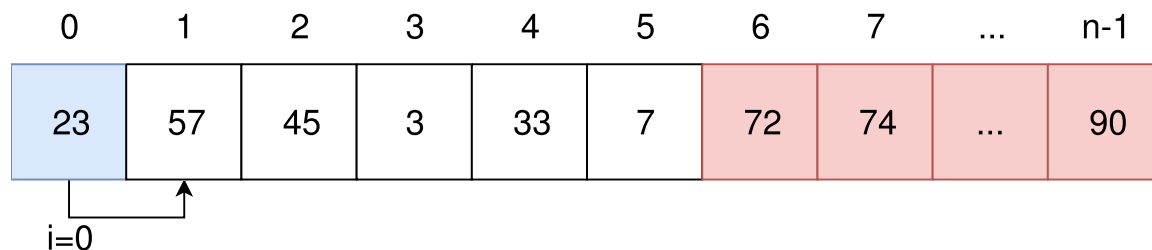
解法

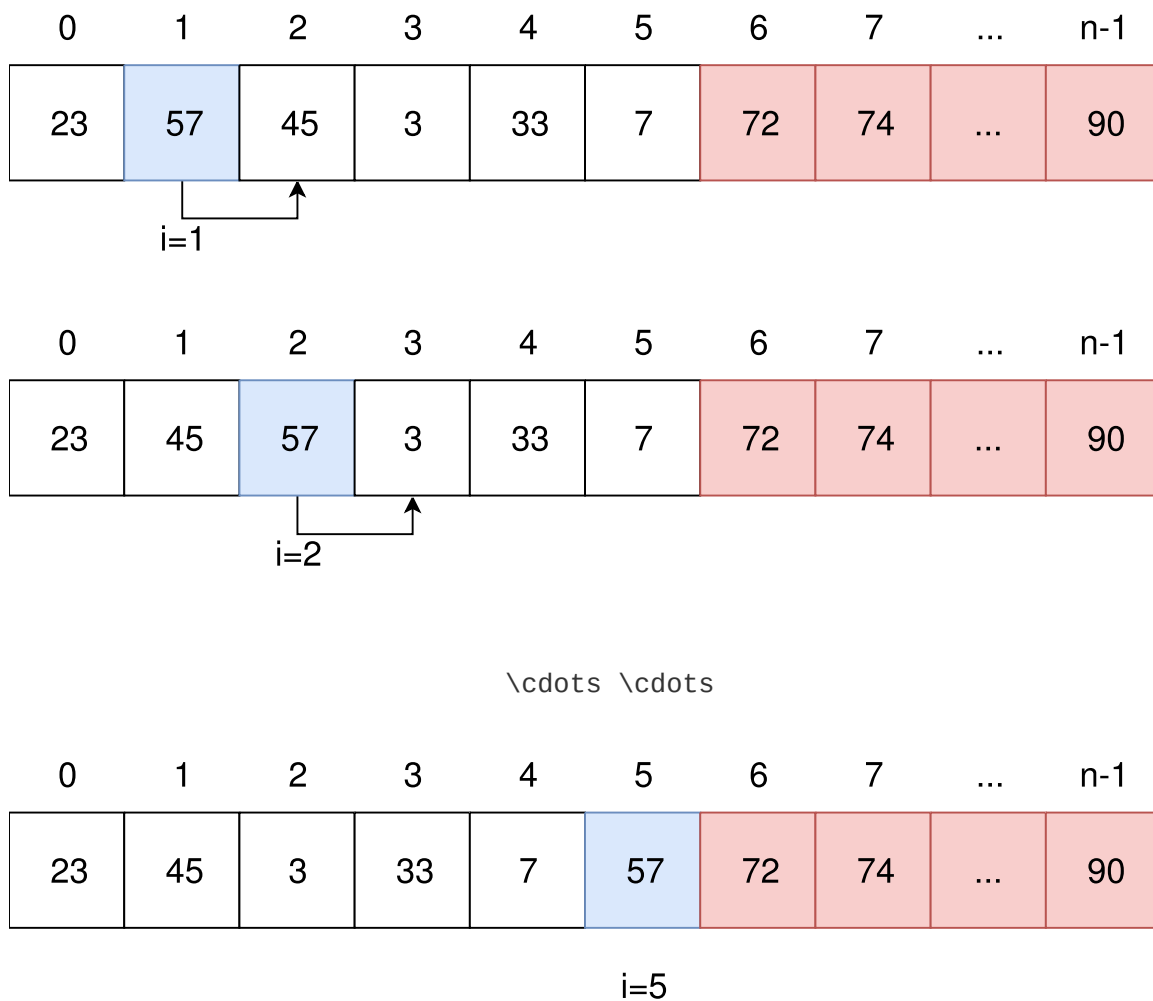
本问题对无序序列 s 升序排序，排序后 s 是从小到大的。

将长度为 n 的序列 s 分为 $left$ 和 $right$ 两个部分，其中 $left$ 是无序部分，范围为 $s[0, k]$ ， $right$ 是有序部分，范围为 $s[k+1, n-1]$ ，其中 $0 \leq k \leq n$ 。初始时 $left$ 范围为 $s[0, n-1]$ ， $right$ 为空。

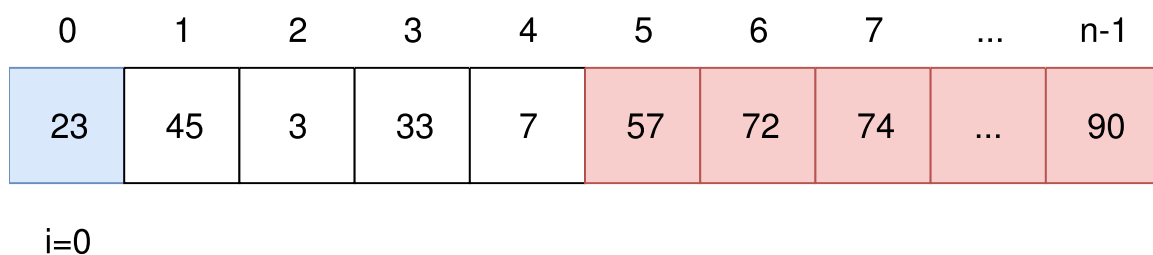
$left$ 从左边第一个元素 $s[i]$ （初始时 $i = 0$ ）开始向右遍历，依次对 $s[i]$ 和 $s[i+1]$ 进行比较，若 $s[i] > s[i+1]$ 则交换两个元素，直到 $i = k$ 为止，完成一次遍历操作。每次遍历会将 $left$ 中的最大元素移动到 $s[0, k]$ 的最右边，之后就可以将 $left$ 的范围缩小为 $s[0, k-1]$ ， $right$ 的范围扩大为 $s[k, n-1]$ 。

例如对于下图中的数组 s ， $left$ 为 $s[0, 5]$ ， $right$ 为 $s[6, n-1]$ 。从 $i = 0$ 开始向右遍历，依次比较 $s[i]$ 和 $s[i+1]$ ，若 $s[i] > s[i+1]$ 则交换两个元素，直到 $i = 5$ 。





然后将 `left` 中的最大值 `s[5] = 57` 合并到 `right` 部分中，再进行新一轮的遍历交换操作。



重复上面的遍历交换操作，从 $i = 0$ 开始向右遍历。这样直到 `left` 部分为空，`right` 部分即为已序数组，算法结束。对于长度为 n 的序列 `s`，每一轮将 `left` 中的最大值移动到 `right`，时间复杂度为 $O(n)$ ，总共需要 n 轮，该算法的时间复杂度为 $O(n^2)$ 。

源码

```
import, lang:"c_cpp"
```


测试

```
import, lang:"c_cpp"
```

QuickSort 快速排序

- [Quick Sort - 快速排序](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Quick Sort - 快速排序

问题

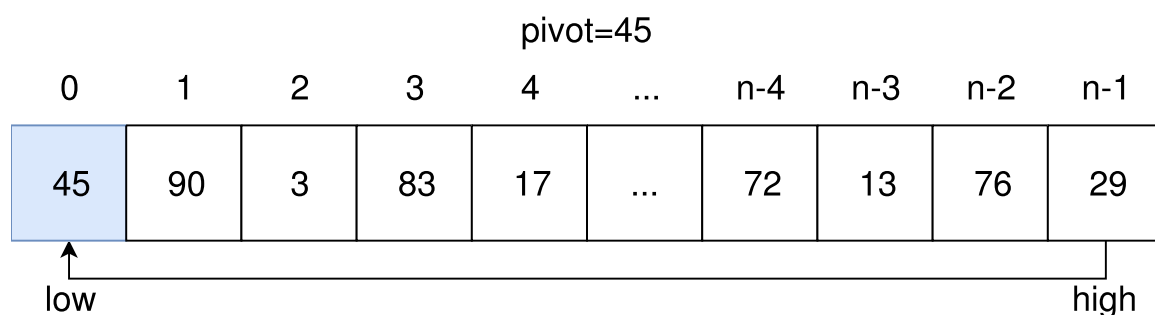
用快速排序对长度为 n 的无序序列 s 进行排序。

解法

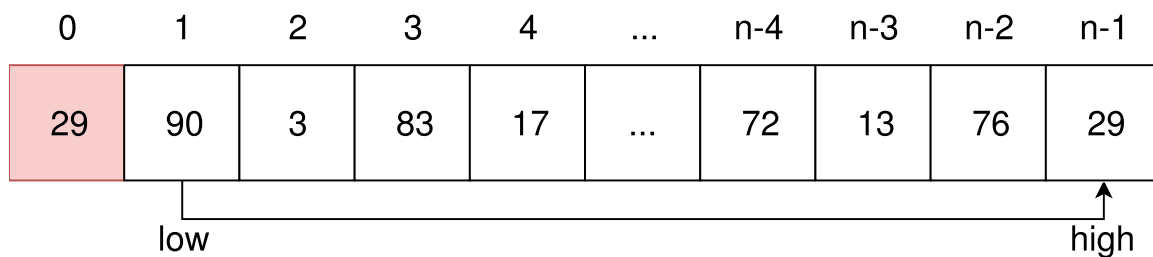
本问题对无序序列 s 进行升序排序，排序后 s 是从小到大的。

将长度为 n 的序列 s ，选取最左边的值作为 pivot ，将剩余部分分为 left 和 right 两个部分， left 和 right 是无序的，且 left 中的所有元素 $\forall x \leq \text{pivot}$ （其中 $x \in \text{left}$ ）， right 中的所有元素 $\forall y \geq \text{pivot}$ （其中 $y \in \text{right}$ ）。

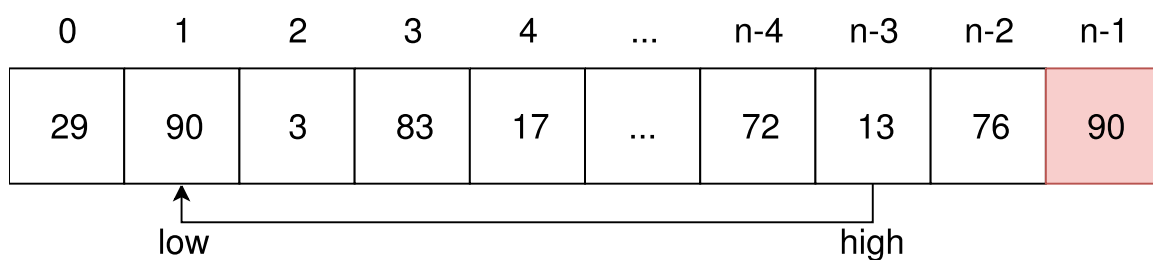
初始时 left 和 right 两个部分都是空的，分别从数组 s 的左右两边向中间推进。例如下图中的数组：



初始时设置 $\text{pivot} = s[0] = 45$ ， $\text{low} = 0$ ， $\text{high} = n-1$ 。从 high 开始，向左搜索到第一个元素 $s[\text{high}] < \text{pivot}$ （ $\text{high} = n-1$ ），该元素不符合 right 的性质，因此将 $s[\text{high}]$ 移动到 $s[\text{low}]$ （ $s[\text{low}] = s[\text{high}]$ ）。



再从 `low` 开始，向右搜索到第一个元素 `s[low] > pivot` (`low = 1`)，该元素不符合 `left` 的性质，因此将 `s[low]` 移动到 `s[high]` (`s[high] = s[low]`)。



重复上面的操作，直到 `low = high`，这时的 `low` 和 `high` 的位置即为 `left` 和 `right` 的中间位置，将 `pivot` 移动到该位置 (`s[low] = pivot`)，就完成了一轮排序。`left` 和 `right` 内部仍然是无序的，把它们也当作一个数组，递归的进行排序即可。

对于长度 `n` 的序列 `s`，每一轮放置所需要的时间为 $O(n)$ ，总共需要 $\log_2 n$ 轮，该算法的时间复杂度为 $O(n \cdot \log_2 n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

MergeSort 归并排序

- Merge Sort - 归并排序
 - 问题
 - 解法
 - 源码
 - 测试

Merge Sort - 归并排序

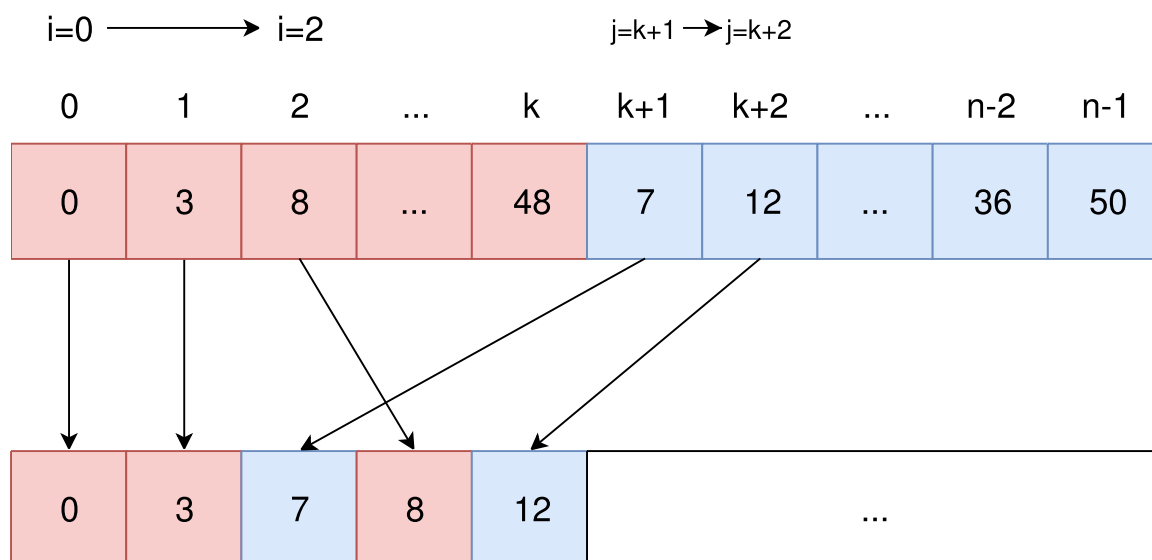
问题

用归并排序对长度为 n 的无序序列 s 进行排序。

解法

本问题对无序序列 s 进行升序排序，排序后 s 是从小到大的。

对于长度为 n 的序列 $s[0, n)$ ，将其从中间分开为 $\text{left}[0, k]$ 和 $\text{right}[k+1, n-1]$ 两个部分 ($0 \leq k < n-1$)，假设 $\text{left}[0, k]$ 和 $\text{right}[k+1, n-1]$ 两个部分已经是升序的，那么只需要将这两个部分进行合并排序即可。设 i 和 j 两个下标分别从 left 和 right 的最左边 ($0 \leq i \leq k$, $k+1 \leq j \leq n-1$) 向右遍历，每次将 $\text{left}[i]$ 和 $\text{right}[j]$ 中较小的值插入新的数组中，即可完成一次合并操作。



递归的将 $\text{left}[0, k]$ 和 $\text{right}[k+1, n-1]$ 分别拆分为更小的 left 和 right 两部分，假定子部分也是升序的，重复上述操作即可得到有序的 $\text{left}[0, k]$ 和 $\text{right}[k+1, n-1]$

。这样递归下去，当某个部分的长度等于1时，可以看作长度为1的有序部分，递归结束。

对于长度 n 的序列 s ，每一轮放置所需要的时间为 $O(n)$ ，总共需要 $\log_2 n$ 轮，该算法的时间复杂度为 $O(n \cdot \log_2 n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

Chapter-2 Search 第2章 搜索

- [Chapter-2 Search](#)
- [第2章 搜索](#)

Chapter-2 Search

第2章 搜索

1. [KnowledgePoint](#) 知识要点
2. [BinarySearch](#) 二分查找法 (折半查找法)
3. [BruteForce](#) 暴力枚举
4. [Recursion](#) 递归
5. [BreadthFirstSearch](#) 广度优先搜索
6. [BidirectionalBreadthSearch](#) 双向广度搜索
7. [AStarSearch](#) A*搜索
8. [DancingLinks](#) 舞蹈链

KnowledgePoint 知识要点

- Knowledge Point - 知识要点

Knowledge Point - 知识要点

一个矩阵的示例 m 如下图：

	0	1	2	3	4
0					
1					
2					

我们称之为 3 行 5 列的矩阵，在计算机程序中一般用二维数组 $m = 5 \times 3$ 表示， $m[\text{col}, \text{row}]$ 表示第 col 行、第 row 列的元素（也可以颠倒过来表示成 $m[\text{row}, \text{col}]$ ，只需要保证统一即可）。比如：

	0	1	2	3	4
0					
1					
2		[1,2]			

	0	1	2	3	4
0				[3,0]	
1					
2					

本书中我们总是将第 1 维作为列 `col`，将第 2 维作为行 `row`。

BinarySearch 二分查找法 (折半查找法)

- [Binary Search - 二分查找法 \(折半查找法\)](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Binary Search - 二分查找法 (折半查找法)

问题

在长度为 n 的有序序列 s 中查找元素 x 的位置。

解法

有序序列 s 可以是升序或降序的，即从小到大或从大到小，本问题假设 s 是升序的。

在长度为 n 的升序序列 s 中想要找出某个元素 x 是否存在，在序列 s 中初始化 $low = 0$, $high = n-1$ 。

当 $low \leq high$ 时，对于范围 $[low, high]$ ，设 $mid = \lfloor \frac{high+low}{2} \rfloor$ (向下取整)，若 $x = s[mid]$ 则 mid 即为所求，算法结束；若 $x < s[mid]$ ，则 x 的位置在子范围 $s[0, mid-1]$ 中，令 $high = mid-1$ ；若 $x > s[mid]$ ，则 x 的位置在子范围 $s[mid+1, n-1]$ 中，令 $low = mid+1$ 。对于缩小的子范围 $[low, high]$ ，重复上述搜索操作，直到找到 $x = s[mid]$ 。若 $low > high$ 时仍然找不到 $x = s[mid]$ ，则序列 s 中不存在 x 。

例如下图中，若 $x = 17 = s[mid]$ ，可以直接找到 $x = s[4]$ ：

0	1	2	3	4	5	6	7	8	9
0	3	7	8	17	26	30	36	48	50
low				mid	high				

若 $x = 5 < s[mid] = 17$ ，则令 $high = 3$ 之后继续搜索：

0	1	2	3	4	5	6	7	8	9
0	3	7	8	17	26	30	36	48	50
low	mid		high						

若 $x = 30 > s[mid] = 17$, 则令 $low = 5$ 之后继续搜索:

0	1	2	3	4	5	6	7	8	9
0	3	7	8	17	26	30	36	48	50
					low		mid		high

对于长度为 n 的序列 s , 每次计算 mid 的时间看作 $O(1)$ 。在最好情况下1次查找就可以找到; 在最坏情况下需要 $\log_2 n$ 次才能找到 x 。该算法的时间复杂度为 $O(\log_2 n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

BruteForce 暴力枚举

- [Brute Force - 暴力枚举](#)
 - [问题](#)
 - [原理](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Brute Force - 暴力枚举

问题

序列 s 有 n 个成员 $[s_1, s_2, \dots, s_n]$ ，每个成员可以选取 $[1, 2, \dots, m]$ 这 m 种植。

例如当 $n = 5$ ， $m = 3$ 时，序列 s 有如下排列组合：

$[1, 1, 1, 1, 1], [1, 1, 1, 1, 2], [1, 1, 1, 1, 3], [1, 1, 1, 2, 1] \dots$

遍历序列 s 的可能排列组合的所有情况。

原理

加法原理：完成一件事情有 n 类方法，每类方法有若干子方法，完成这件事需要且只需要 n 类方法中的一类方法中的一个子方法。第 1 类方法有 m_1 种子方法，第 2 类方法有 m_2 种子方法， \dots ，第 n 类方法有 m_n 种子方法。则完成这件事共有 $m_1 + m_2 + \dots + m_n$ 种方法。

乘法原理：完成一件事情需要 n 个步骤，每个步骤有若干子方法，完成这件事情需要 n 个步骤都完成，每个步骤需要且只需要选择一种方法。第 1 步有 m_1 种子方法，第 2 步有 m_2 种子方法， \dots ，第 n 步有 m_n 种子方法。则完成这件事共有 $m_1 \times m_2 \times \dots \times m_n$ 种方法。

解法

通过 `for` 循环枚举出序列 s 中的所有可能。

例如对于序列 $[s_1, s_2, s_3, s_4]$ ，其中每个元素的取值范围是 $[0, m]$ 。如果把该序列看作一个正整数，从 0000 依次数到 9999 即为全部的排列组合。

对于成员数量为 n ，每个成员有 m 种值的序列 s ，遍历所有排列组合的时间复杂度 $O(n^m)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

Recursion 递归

- [Recursion - 递归](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Recursion - 递归

问题

序列 s 有 n 个成员 $[x_1, x_2, \dots, x_n]$ ，每个成员可以选取 $[1, 2, \dots, m]$ 这 m 种植。例如当 $n = 5$ ， $m = 3$ 时，序列 s 有如下排列组合：

$[1, 1, 1, 1, 1], [1, 1, 1, 1, 2], [1, 1, 1, 1, 3], [1, 1, 1, 2, 1] \dots$

求 s 的所有排列组合。（与本节的<BruteForce>问题一样）

解法

<BruteForce>存在一个问题，外围for循环的数量是固定的，无法改变。因此我们用递归来解决这个问题。假设序列 s 从长度 0 增长到 n 。在长度为 i 的基础上，我们找出序列 s 增加一个元素，成为长度为 $i+1$ 时的所有可能的排列组合（其中 $0 \leq i < n$ ）。初始化时令序列为空 $s = []$ 。

(1) 将序列 s 的长度增加到 1 ，第 1 个元素（唯一的元素） x_1 有 m 种选择，即长度为 1 的序列 s 有 m 个排列组合：

$[1_1]$

$[2_1]$

\dots

$[m_1]$

(2) 将序列 s 的长度增加到 2，得到数组 $s = [x_1, x_2]$ ，元素 x_2 的选择可以看作在第 (1) 轮的每个选择的基础上继续选择。对于 $[1_1]$ 可以得到 m 个排列组合：

$$[1_1, 1_2]$$

$$[1_1, 2_1]$$

$$\vdots$$

$$[1_1, m_1]$$

第 (2) 轮操作后共有 m^2 个排列组合。重复 n 次这样的操作，可以得到 m^n 个排列组合。

实际编写代码中，在递归方程中传入一个参数 $prev \in [0, n)$ ， $prev$ 从 0 开始，序列 s 中的成员 $x_{\{prev\}}$ 可以取值 $i \in [1, m]$ ，然后 $prev = prev + 1$ ，继续考虑序列 s 中的下一个成员 $x_{\{prev+1\}}$ 。这样直到当 n 个成员都选择了一个值时，即产生序列 s 的一种排列组合。通过递归可以退回上一个函数栈，从而让每个成员 $x_{\{prev\}}$ 都可以重新选择。

对于成员数量为 n ，每个成员有 m 种值的序列 s ，遍历所有排列组合的时间复杂度 $O(m^n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

BreadthFirstSearch 广度优先搜索

- [Breadth First Search - 广度优先搜索](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Breadth First Search - 广度优先搜索

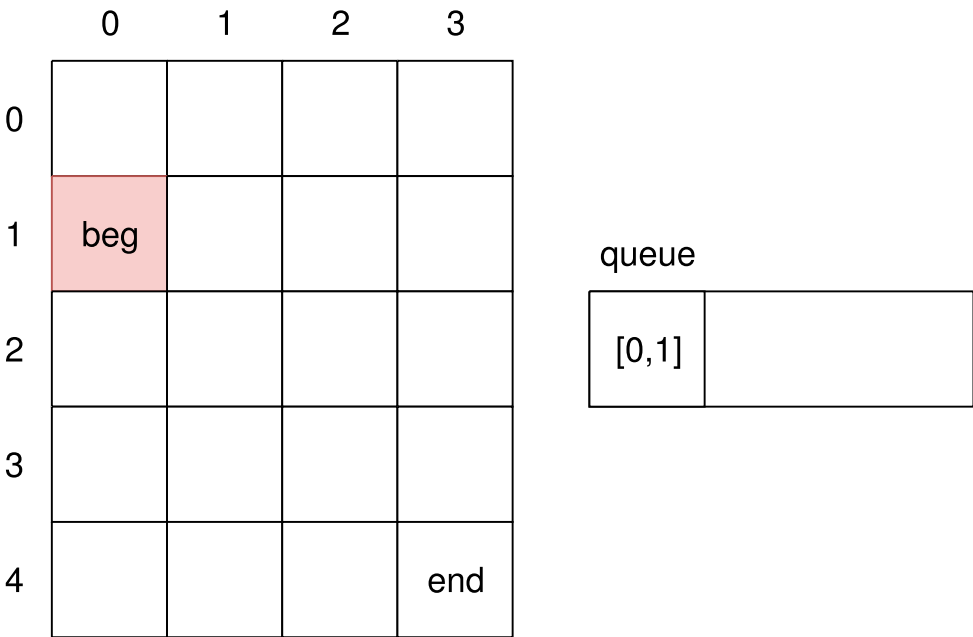
问题

在 $m \times n$ 的二维方格图 s 中从 beg 点移动到 end 点。

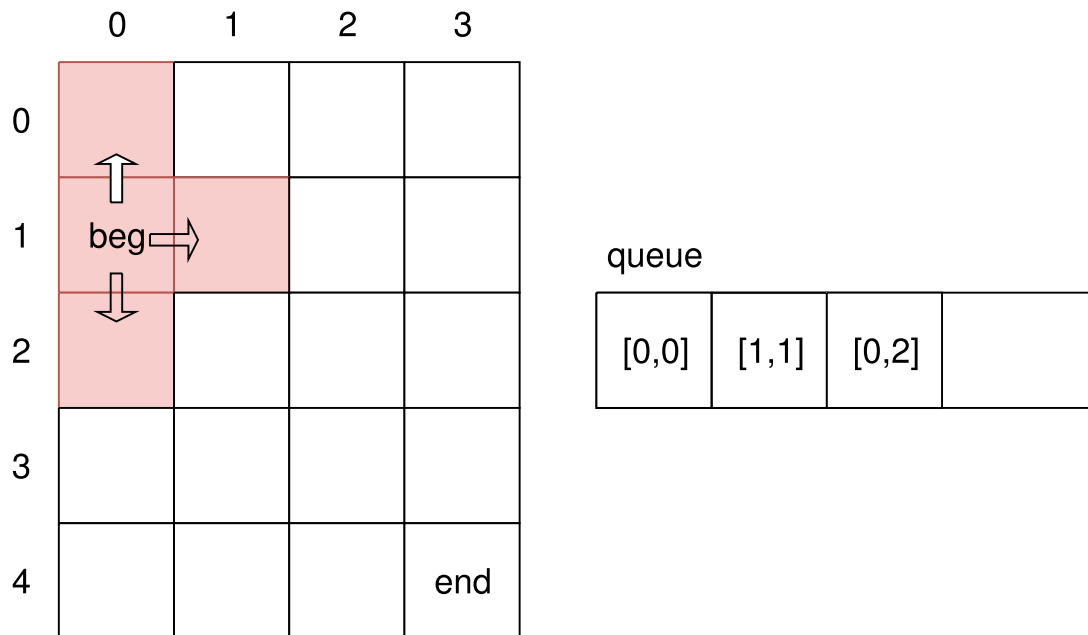
解法

广度优先搜索是优先搜索二维方格图 s 中每个节点的相邻节点，与之相对的深度优先搜索则会沿着节点的一个相邻节点试图走到最远。

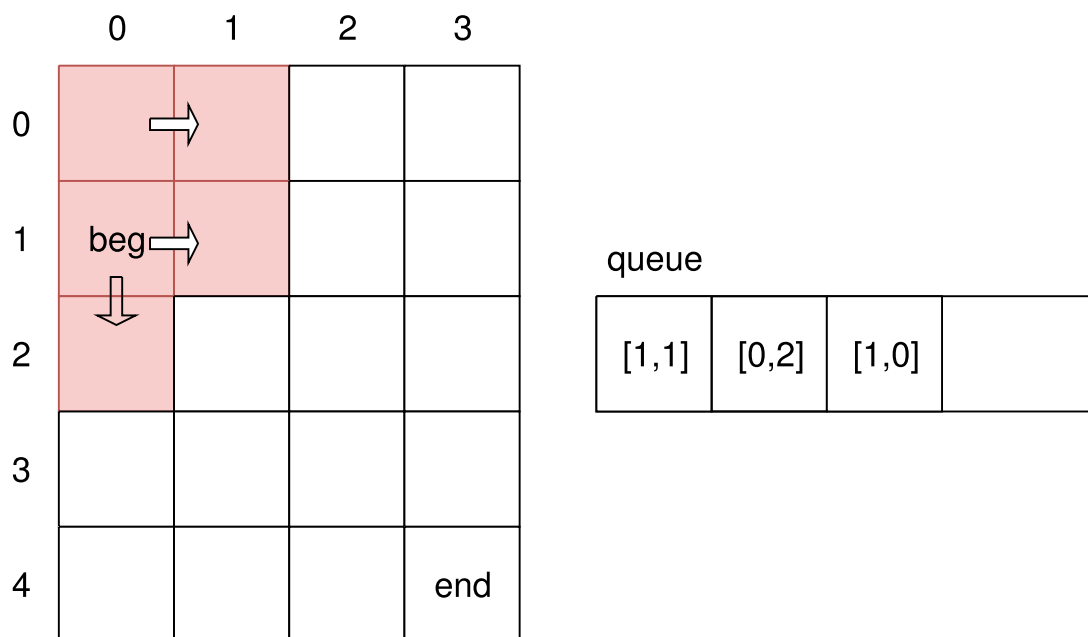
例如在下面这个 4×5 二维方格 s 中从 $beg = [0,1]$ 移动到 $end = [3,4]$ 。初始时将起点 beg 加入等待搜索的队列 $queue$ 中，之后每次从 $queue$ 中取出头节点 e ，访问 e 四周从未被访问的邻节点，并将邻节点加入 $queue$ 中。将每个节点加入 $queue$ 之前将其染为红色，避免重复访问。



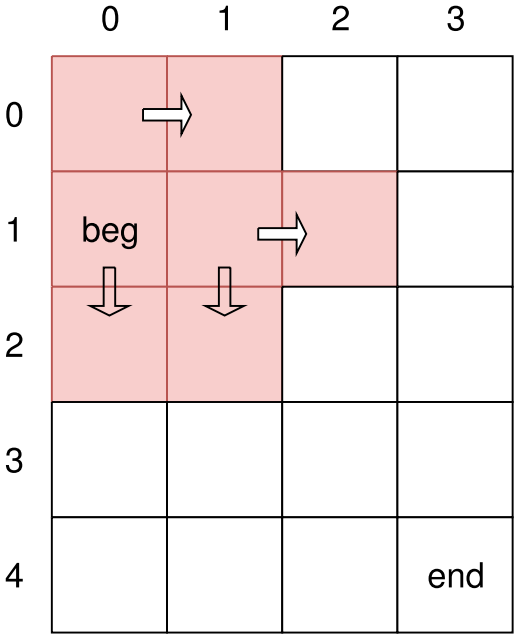
(1) 初始时将 $beg = [0, 1]$ 染红并加入 $queue$ ；



(2) 从 $queue$ 中取出头节点 $[0, 1]$ ，因 $[0, 1] \neq end$ ，将其四周未被染红的节点 $[0, 0]$ 、 $[1, 1]$ 、 $[0, 2]$ 染红并加入 $queue$ ；

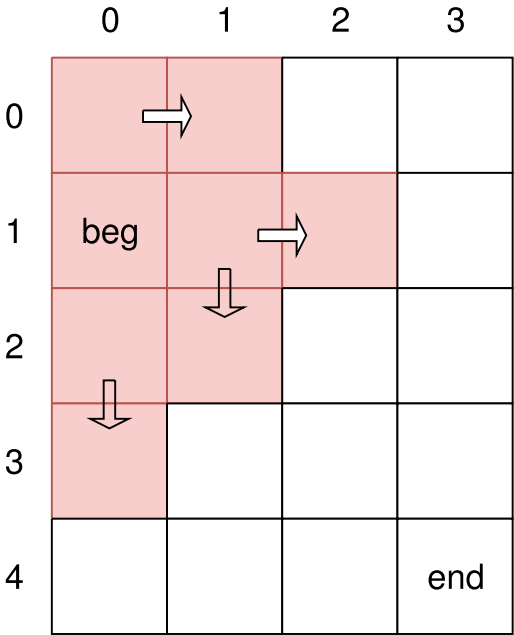


(3) 从 $queue$ 中取出头节点 $[0, 0]$ ，因 $[0, 0] \neq end$ ，将其四周未被染红的节点 $[1, 0]$ 染红并加入 $queue$ ；



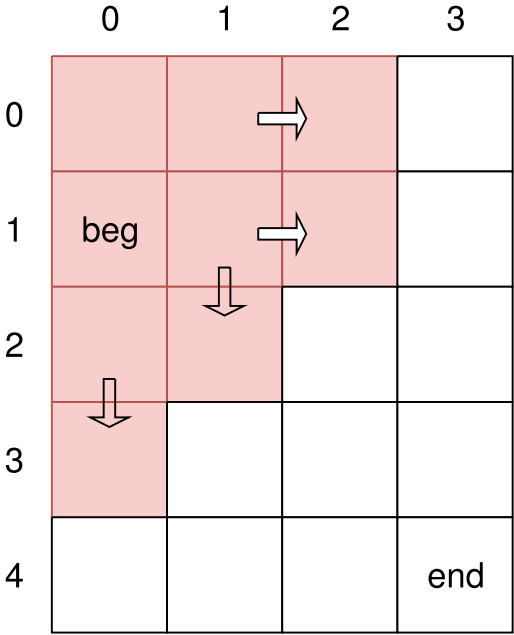
queue

[0,2]	[1,0]	[2,1]	[1,2]	
-------	-------	-------	-------	--



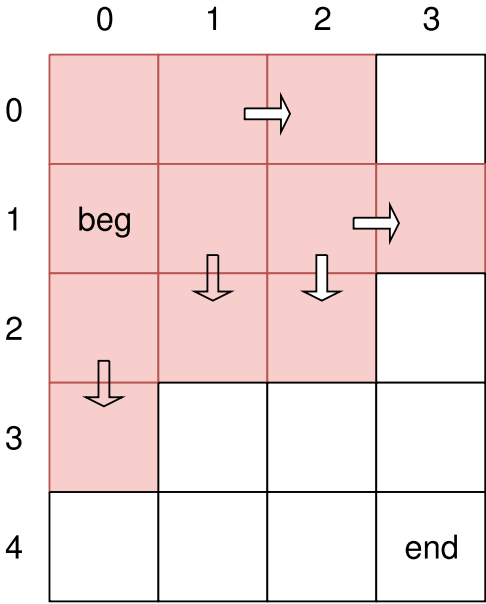
queue

[1,0]	[2,1]	[1,2]	[0,3]	
-------	-------	-------	-------	--



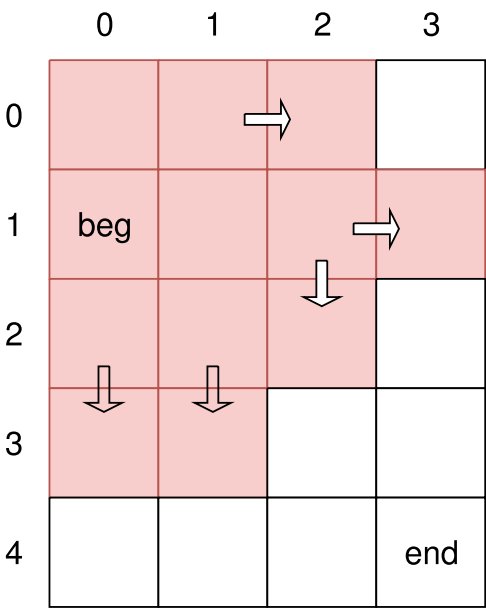
queue

[2,1]	[1,2]	[0,3]	[2,0]	
-------	-------	-------	-------	--



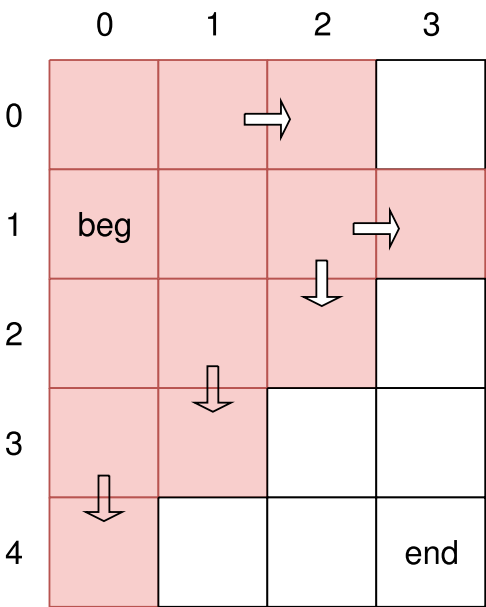
queue

[1,2]	[0,3]	[2,0]	[2,2]	[3,1]	
-------	-------	-------	-------	-------	--



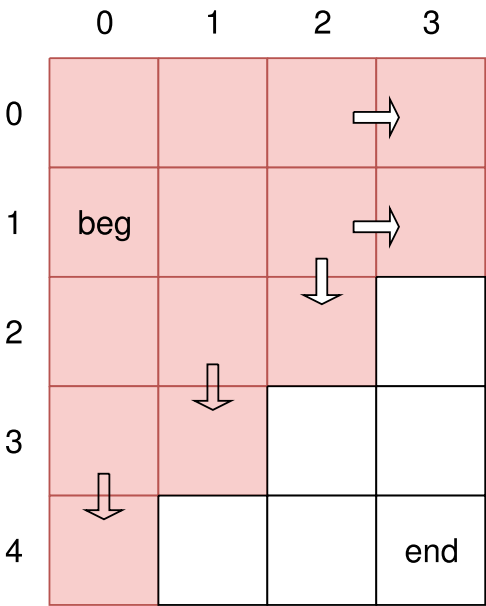
queue

[0,3]	[2,0]	[2,2]	[3,1]	[1,3]	
-------	-------	-------	-------	-------	--



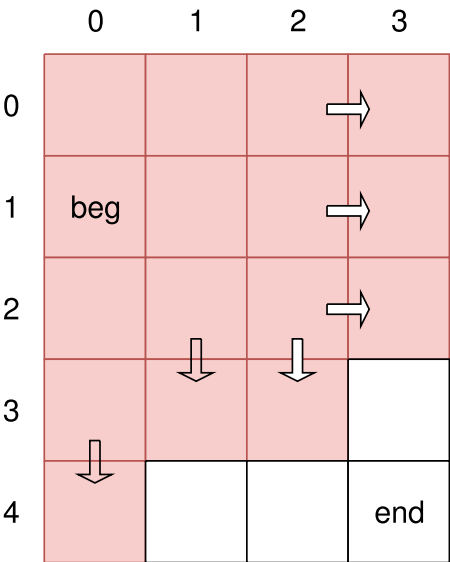
queue

[2,0]	[2,2]	[3,1]	[1,3]	[0,4]	
-------	-------	-------	-------	-------	--



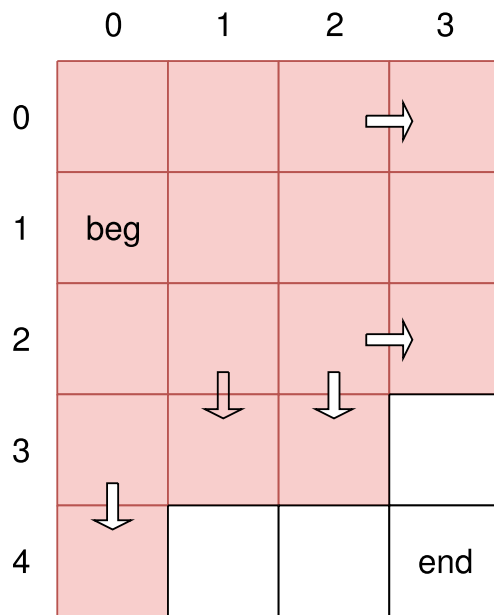
queue

[2,2]	[3,1]	[1,3]	[0,4]	[3,0]	
-------	-------	-------	-------	-------	--



queue

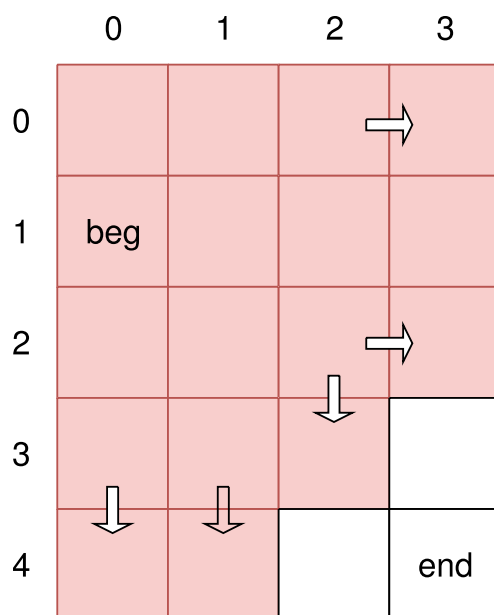
[3,1]	[1,3]	[0,4]	[3,0]	[2,3]	[3,2]	
-------	-------	-------	-------	-------	-------	--



queue

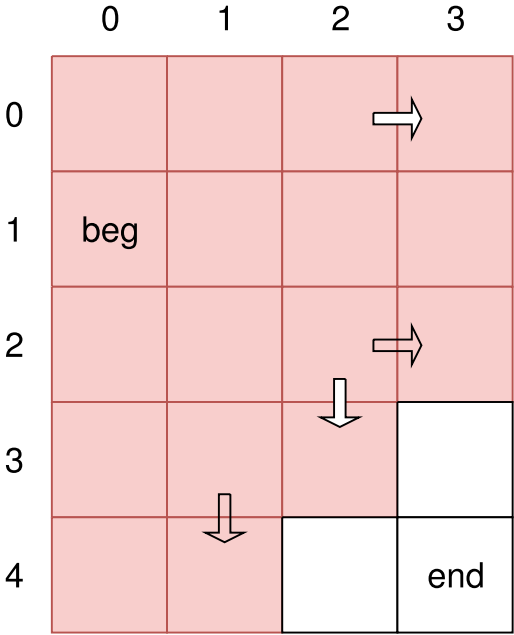
[1,3]	[0,4]	[3,0]	[2,3]	[3,2]	
-------	-------	-------	-------	-------	--

(4) 从 queue 中取出头节点 $[3,1]$ ，因 $[3,1] \neq \text{end}$ ，其四周的节点都已经被染红，因此不做任何操作；



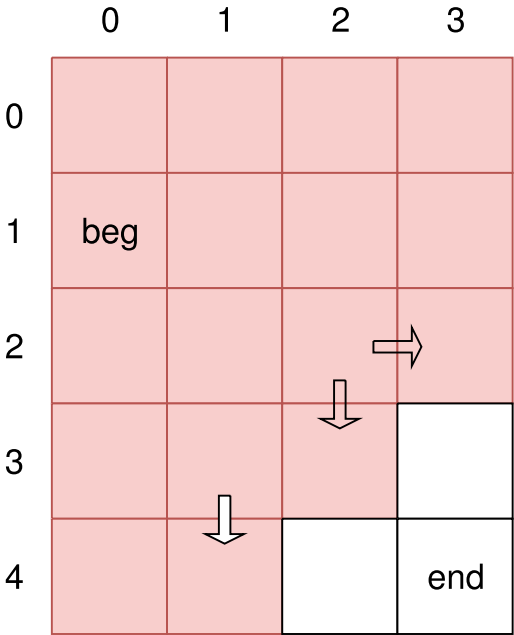
queue

[0,4]	[3,0]	[2,3]	[3,2]	[1,4]	
-------	-------	-------	-------	-------	--



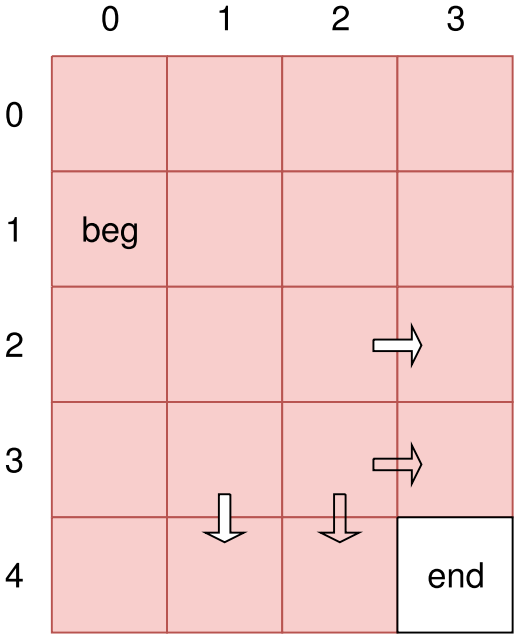
queue

[3,0]	[2,3]	[3,2]	[1,4]	
-------	-------	-------	-------	--



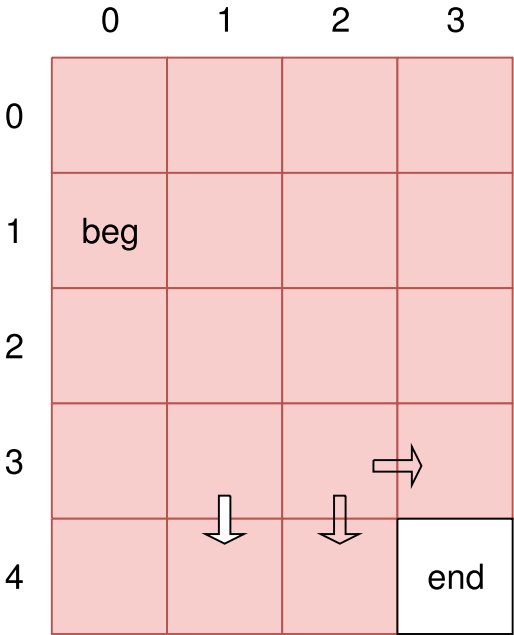
queue

[2,3]	[3,2]	[1,4]	
-------	-------	-------	--



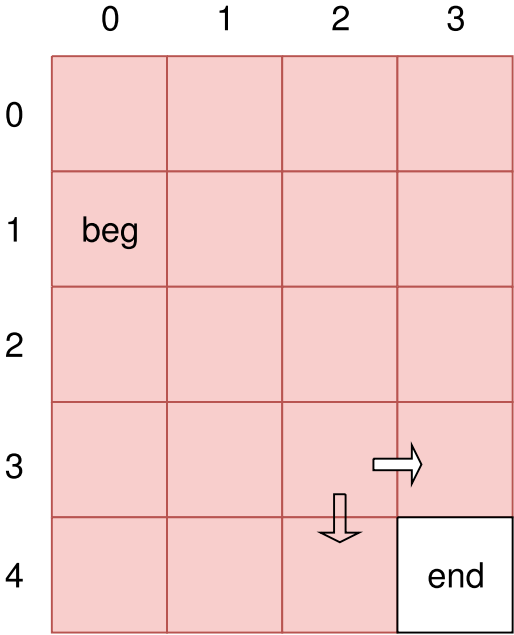
queue

[3,2]	[1,4]	[2,4]	[3,3]	
-------	-------	-------	-------	--



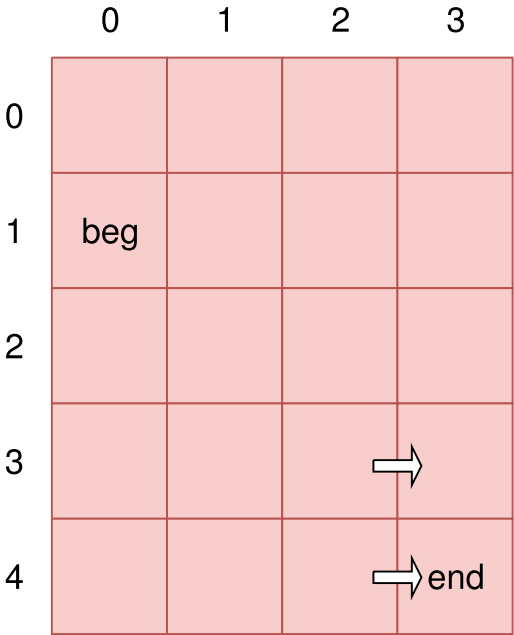
queue

[1,4]	[2,4]	[3,3]	
-------	-------	-------	--



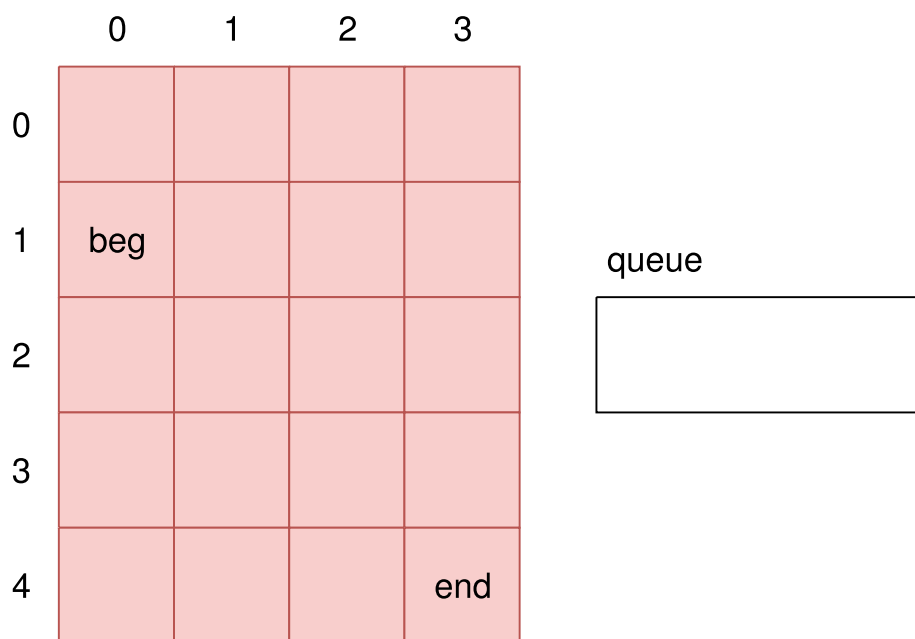
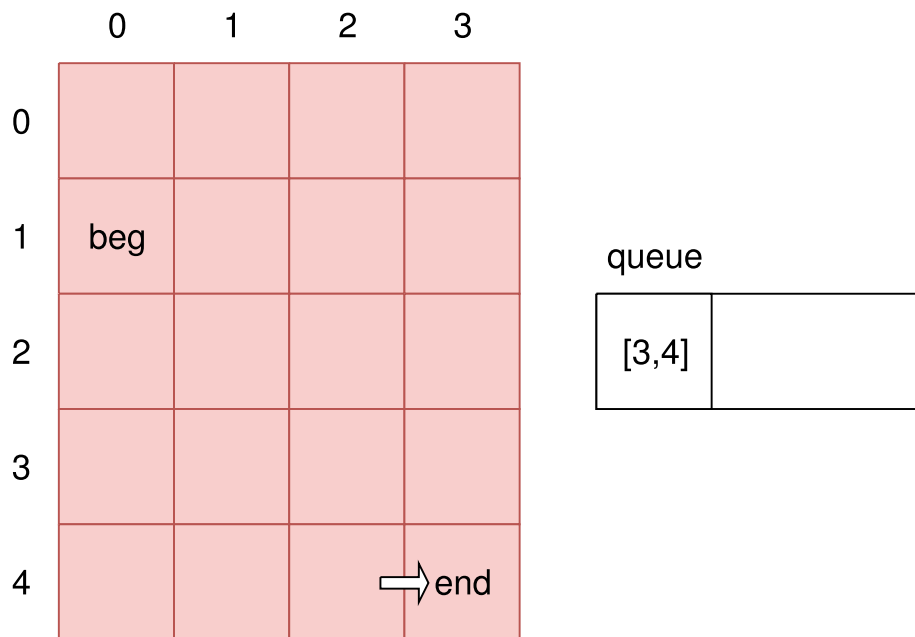
queue

[2,4]	[3,3]	
-------	-------	--



queue

[3,3]	[3,4]	
-------	-------	--



(5) 从 queue 中取出头节点 $[3,4]$ ，因 $[3,4] = \text{end}$ ，算法结束；

上列图中的队列 queue 中，左边为头部，右边为尾部，新访问的节点插入队列尾部，每次从队列中取出头节点 e。如果需要额外的获取从 beg 点到 end 点的完整路径，需要在遍历时标记每个节点的上一个点，即“父节点”，最终可以从 end 通过父节点指针逆向的找到一条回到 beg 点的路径。

该算法下时间复杂度为 $O(m \times n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

BidirectionalBreadthSearch 双向广度搜索

- [Bidirectional Breadth Search - 双向广度搜索](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Bidirectional Breadth Search - 双向广度搜索

问题

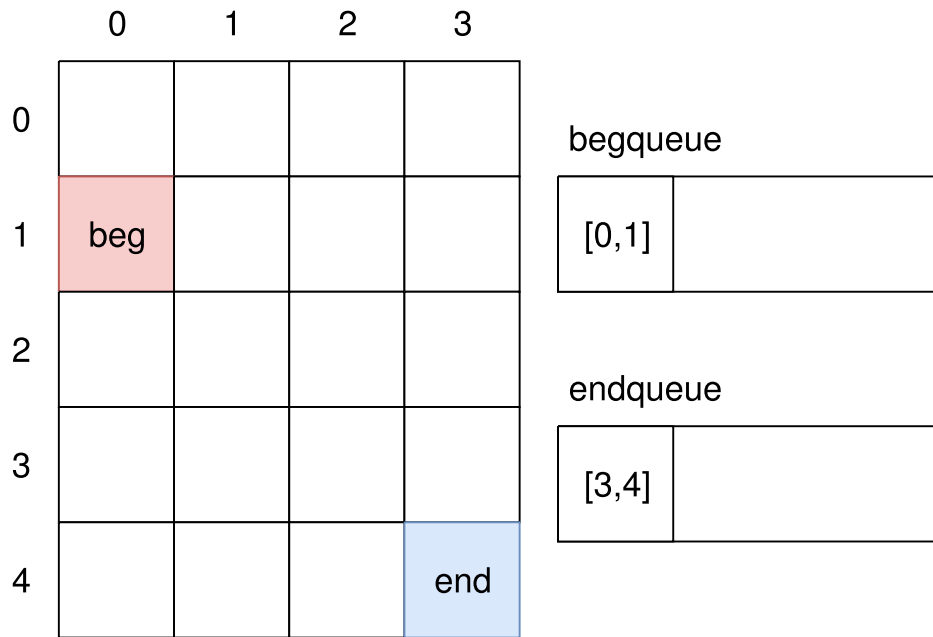
在 $m \times n$ 的二维方格图 s 中用双向广度搜索从 beg 点移动到 end 点。

解法

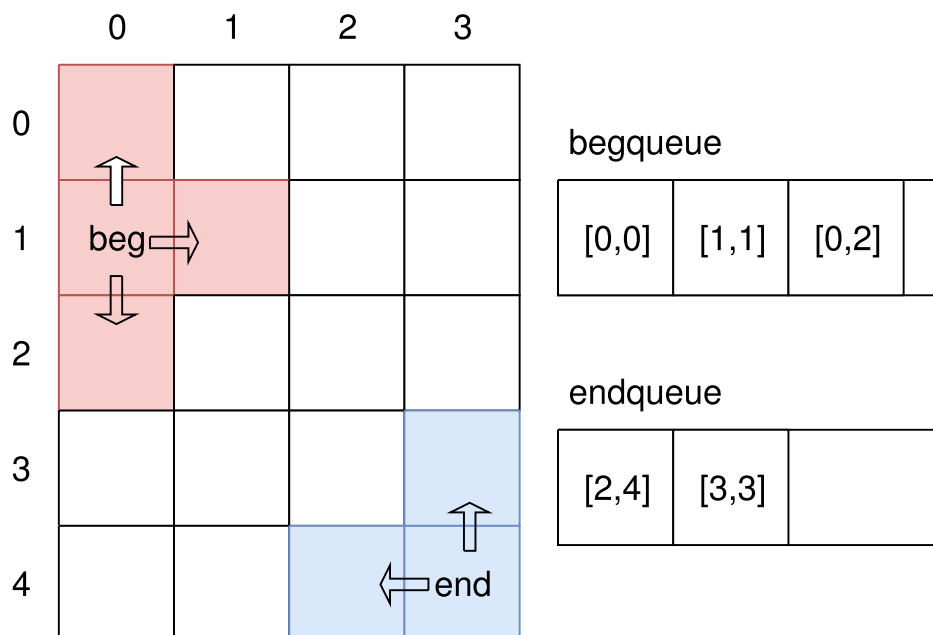
双向广度优先搜索是在广度优先搜索基础上的一个变种，搜索速度更快。该算法从 beg 和 end 两个点开始，同时进行广度优先搜索，两边的点在某一处相遇，即可得到一条从 beg 到 end 的路径。

初始时将 beg 和 end 分别加入两个队列 $begqueue$ 和 $endqueue$ 中。每次分别从 $begqueue$ 和 $endqueue$ 队列中取出节点 x 和 y 进行访问，在节点加入 $begqueue$ 之前将其染成红色，加入 $endqueue$ 之前其染成绿色。若 x 取出后发现已被染成绿色，说明 x 被 $endqueue$ 访问过，或 y 取出后发现其已被染成红色，说明 y 被 $begqueue$ 访问过。说明两个队列在此处相遇，算法结束。

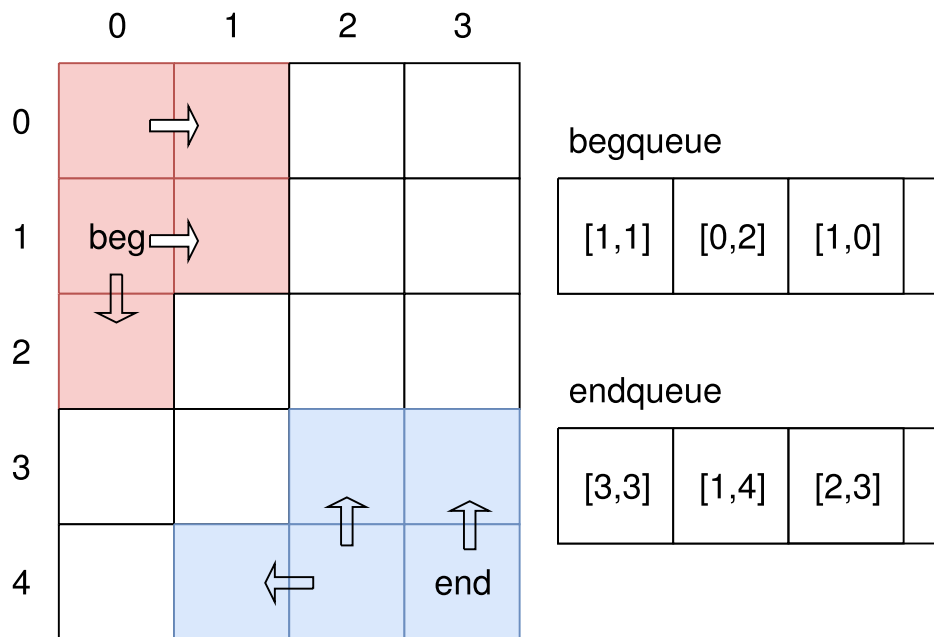
在下面这个 $m = 4$, $n = 5$ 的二维方格 s 中，从 $beg = [0, 1]$ 移动到 $end = [3, 4]$ 的过程如下：



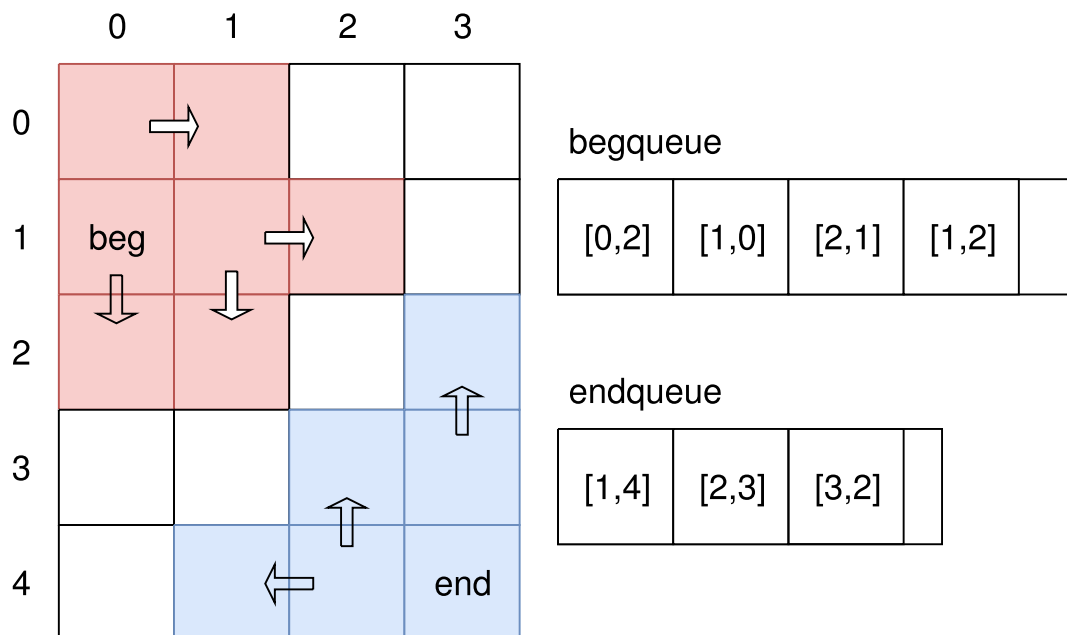
(1) 初始时，将 $beg = [0,1]$ 染红并加入 $begqueue$ 中，将 $end = [3,4]$ 染绿并加入 $endqueue$ ；



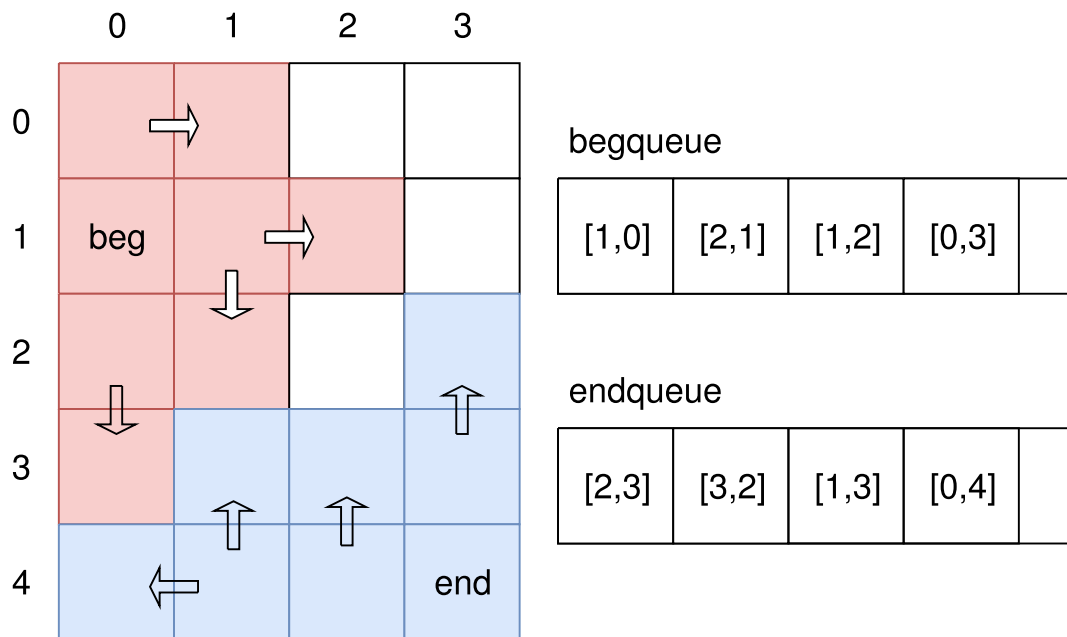
(2) 从 $begqueue$ 中取出头节点 $[0,1]$ ，将其四周围未被染色的邻节点 $[0,0]$ 、 $[1,1]$ 、 $[0,2]$ 染红并加入 $begqueue$ 中。从 $endqueue$ 中取出头节点 $[3,4]$ ，将其四周围未被染色的邻节点 $[2,4]$ 、 $[3,3]$ 染红并加入 $endqueue$ 中；



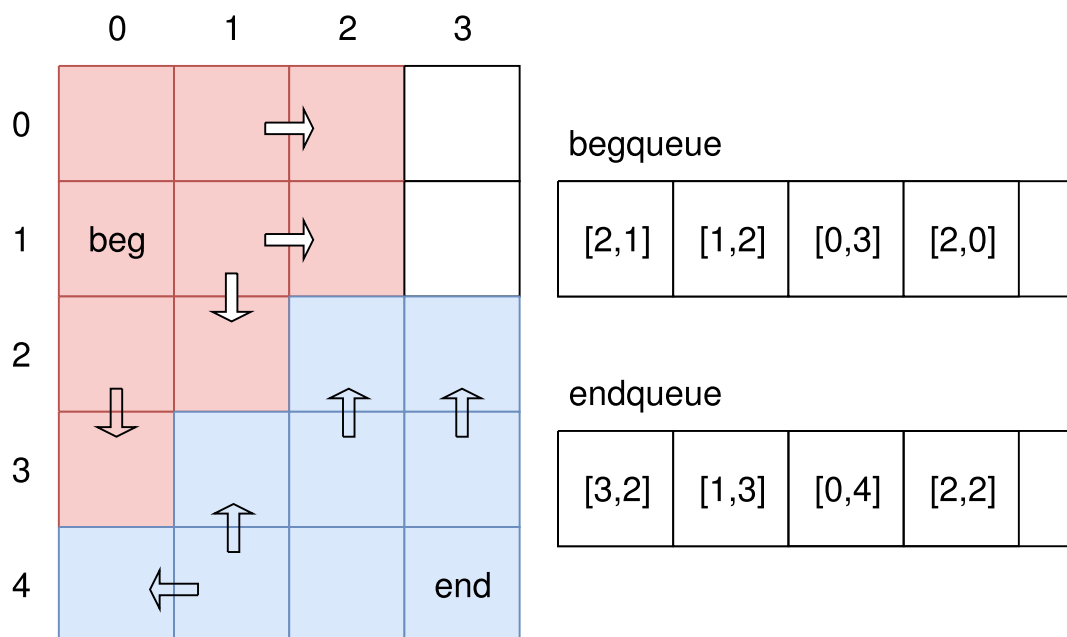
(3) 从 begqueue 中取出头节点 $[0,0]$ ，将其四周末被染色的邻节点 $[1,0]$ 染红并加入 begqueue 中。从 endqueue 中取出头节点 $[2,4]$ ，将其四周末被染色的邻节点 $[1,4]$ 、 $[2,3]$ 染红并加入 endqueue 中；



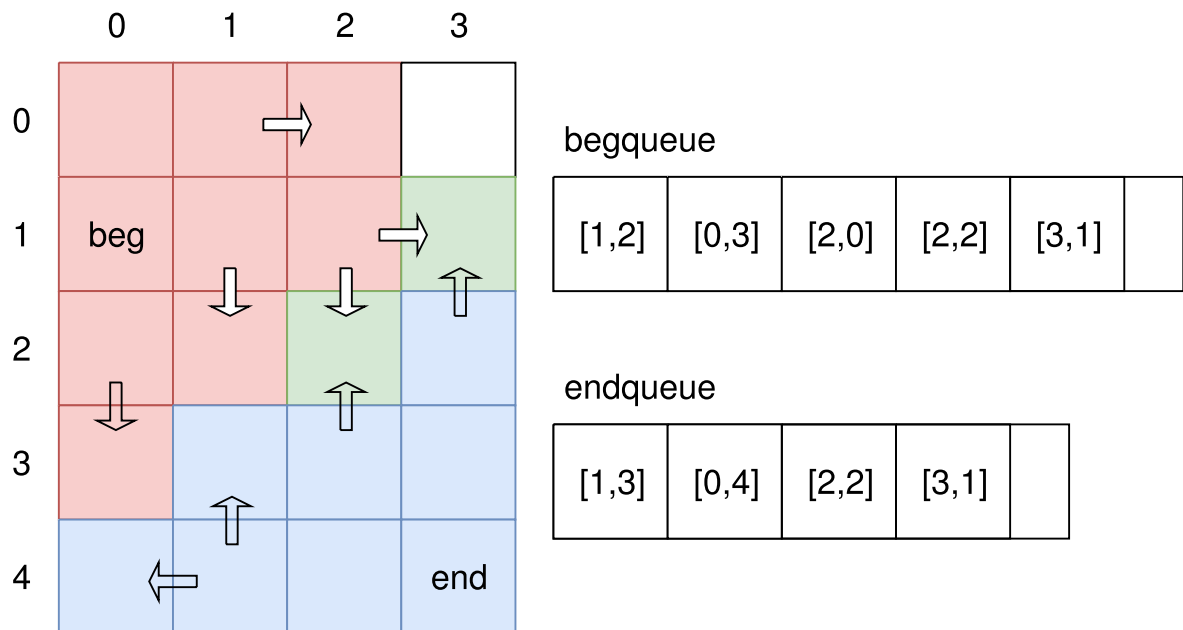
(4) 从 begqueue 中取出头节点 $[1,1]$ ，将其四周末被染色的邻节点 $[2,1]$ 、 $[1,2]$ 染红并加入 begqueue 中。从 endqueue 中取出头节点 $[3,3]$ ，将其四周末被染色的邻节点 $[3,2]$ 染红并加入 endqueue 中；



(5) 从 begqueue 中取出头节点 $[0,2]$ ，将其四周围未被染色的邻节点 $[0,3]$ 染红并加入 begqueue 中。从 endqueue 中取出头节点 $[1,4]$ ，将其四周围未被染色的邻节点 $[1,3]$ 、 $[0,4]$ 染红并加入 endqueue 中；

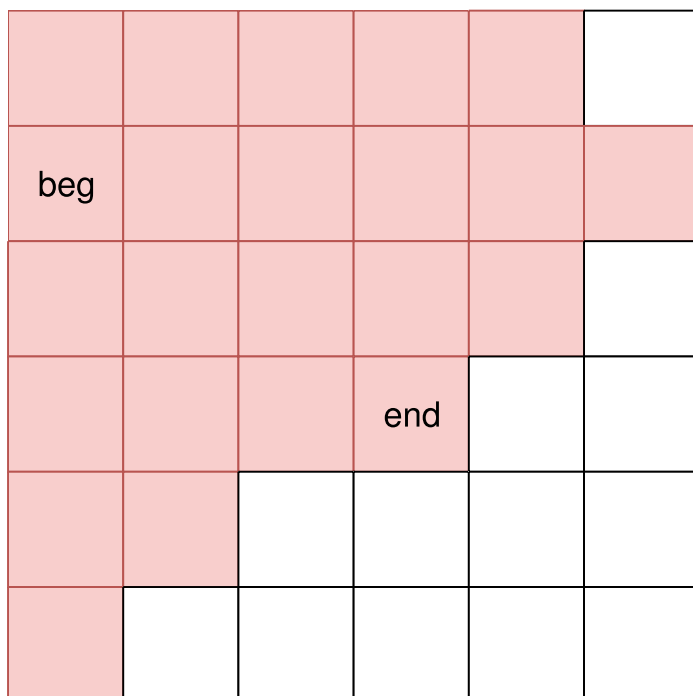


(6) 从 begqueue 中取出头节点 $[1,0]$ ，将其四周围未被染色的邻节点 $[2,0]$ 染红并加入 begqueue 中。从 endqueue 中取出头节点 $[2,3]$ ，将其四周围未被染色的邻节点 $[2,2]$ 染红并加入 endqueue 中；

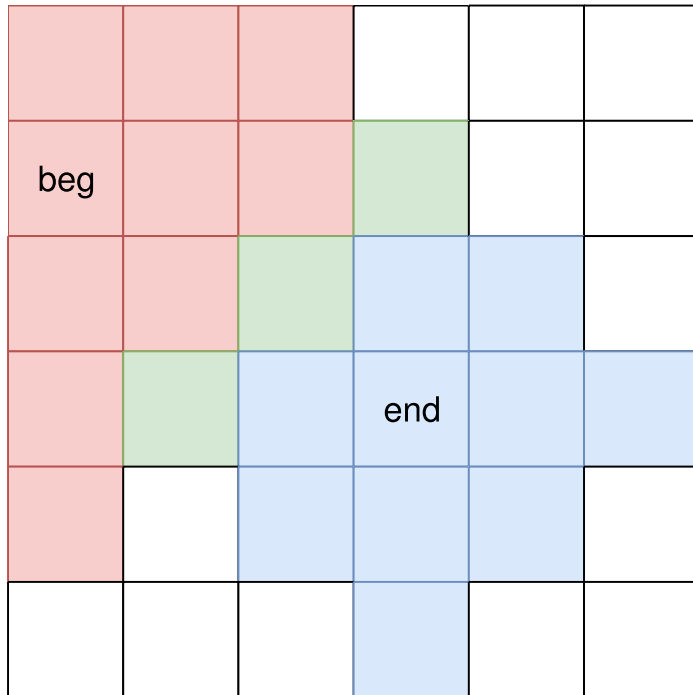


(7) 从 begqueue 中取出头节点 $[2,1]$ ，将其邻节点 $[2,2]$ 已经被染绿，说明该节点已经被加入 endqueue 中，或已经被 endqueue 访问过了。因此 $[2,2]$ 为 begqueue 和 endqueue 相遇的位置，算法结束；

对于二维方格 s ，广度优先搜索从 beg 点遍历到 end 点的过程一般是从 beg 向四周发散开，一直到达 end 点：



而双向广度优先搜索则是从 beg 和 end 两个点分别发散开，在中间相遇：



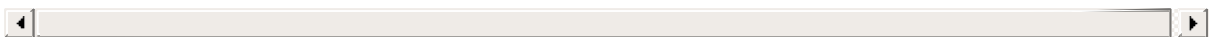
双向广度搜索的时间复杂度与广度优先搜索一样，也是 $O(m \times n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



AStarSearch A*搜索

- [A Star Search - A*搜索](#)
 - [问题](#)
 - [解法](#)
 - [八数码问题](#)
 - [源码](#)
 - [测试](#)

A Star Search - A*搜索

问题

对于 3×3 的矩阵

```
\begin{bmatrix}
2 & 8 & 1 \backslash \\
3 & 7 & x \backslash \\
6 & 4 & 5
\end{bmatrix}
```

x 点可以与上下左右的相邻点交换位置，除此之外不能随意改变位置，将该矩阵变成

```
\begin{bmatrix}
1 & 2 & 3 \backslash \\
4 & 5 & 6 \backslash \\
7 & 8 & x
\end{bmatrix}
```

求最少变换次数以及变化经过，若将矩阵的初始状态看作起点 beg ，最终状态看作终点 end ，即搜索 beg 到 end 的最短路径。

本问题的原型是“八数码问题”。

解法

与之前问题不同，本问题将每种矩阵状态看作一个节点，是一种时间上的状态搜索。本文用A*搜索来解决该问题，A*算法是一种启发式搜索，与DFS和BFS这种无差别搜索不同，A*算法设置一个评价函数 $f(x)$ 来计算节点 x 的搜索代价（到目标的距离），优先搜索那些离目标最近的点，从而提

高搜索效率。

A*算法的评价函数 $f(x) = g(x) + h(x)$ ，其中 x 是节点， $f(x)$ 表示 x 点到 end 的评价距离， $g(x)$ 表示从 beg 节点到 x 节点的实际最短距离， $h(x)$ 表示从 x 点到 end 节点的估算距离。在A*算法的等待队列中，总是优先选取 $f(x)$ 最小的点进行搜索。

在本问题中矩阵节点 x 的估算距离为 x 与 end 在对应位置 $[i, j]$ （其中 $i, j \in [1, 3]$ ）上不同数字的数量之和：

$$\begin{aligned} h(x) &= \sum_{i=1}^3 \sum_{j=1}^3 k_{i,j} \\ k_{i,j} &= \begin{cases} 1 & \{ x_{i,j} \neq end_{i,j} \} \\ 0 & \{ x_{i,j} = end_{i,j} \} \end{cases} \end{aligned}$$

对于下面的矩阵， $h(a) = 6$ ， $h(b) = 7$ ：

\$\$

a

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & x \end{bmatrix}$$

□

b

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & x & 5 \\ 7 & 8 & 6 \end{bmatrix}$$

□

c

$$\begin{bmatrix} 1 & x & 3 \\ 5 & 2 & 6 \\ 4 & 7 & 8 \end{bmatrix}$$

\$\$

与之前的问题不同，本问题中的节点是一种矩阵状态。之前的解法中我们用染色的方式来标记节点是否被访问过，编码实现时可以用数组下标来标记节点 i 的颜色。而矩阵状态是无法作为数组下标的，不过我们可以用哈希表来记录矩阵状态 x 是否被访问过，以及从 beg 节点到达 x 的节点距离。矩阵可以通过下面这两种方式分别映射为字符串或数字（`string`和`int`都可以作为哈希表键值）：

\$\$

```
\begin{bmatrix}
```

```
1 & 2 & 3 \
```

```
4 & x & 5 \
```

```
7 & 8 & 6
```

```
\end{bmatrix}
```

```
\rightarrow
```

```
string("1234x5786")
```

```
\quad
```

```
\begin{bmatrix}
```

```
1 & 2 & 3 \
```

```
4 & x & 5 \
```

```
7 & 8 & 6
```

```
\end{bmatrix}
```

```
\rightarrow
```

```
int(123495786)
```

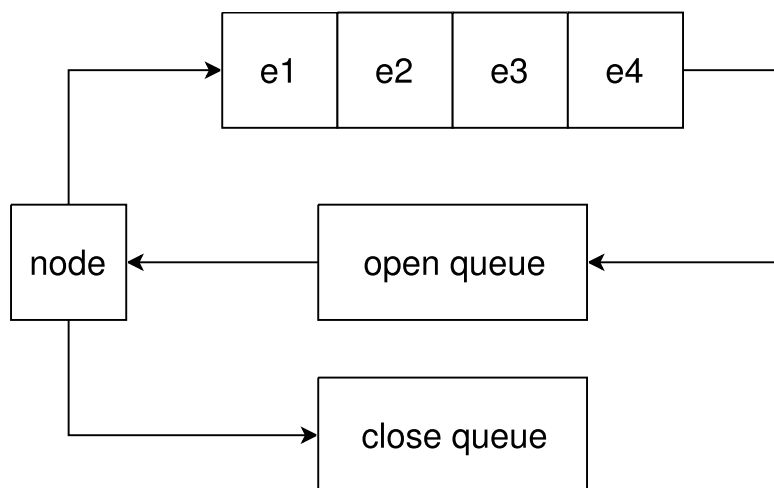
\$\$

设置 `open` 表、`close` 表和 `g` 分数表。`g` 分数表是一个哈希表 $x \rightarrow g(x)$ ，用来存储每个节点的实际距离 $g(x)$ 。`open`表是一个优先队列，与之前搜索算法中的队列功能相同，用于管理等待访问的节点，但是我们需要从`open`表中总是优先取出可能离 `end` 最近的节点，因此其优先级为评价距离 $f(x)$ 。`close`表是一个节点 x 的集合，用于查询所有已经访问过的节点。

初始时我们将 `beg` 节点插入 `open` 表和 `close` 表中，令 $g(beg) = 0$ 。

每一次搜索中，从 `open` 表中取出评价距离 f 最小的节点 `node`，若 `node = end` 则算法结束；否则将 `node` 插入 `close` 表中（也可称为染色，染色的、属于 `close` 表中的节点都是不能再被访问的），该矩阵中的 ‘`x`’ 与上下左右四个数字交换位置，得到新的四个节点 `e_1`、`e_2`、`e_3`、`e_4`，若他们不在 `close` 表中，将其插入 `open` 表和 `close` 表中。

在维护 `open` 中节点的优先级时需要使用 $g(x)$ ，因为 $f(x) = g(x) + h(x)$ 。



当搜索到 open 表中没有节点可以访问时，则说明 beg 节点永远无法到达 end 节点，两个矩阵状态无法转换。更复杂一些的情况，在 beg 可以到达end的基础上，需要求出从 beg 到 end 的路径，这时我们可以把 close 表改为哈希表 $x \rightarrow \text{from}(x)$ ，用来存储节点 x 及其父节点 from ，最后从 end 节点反向，通过查找 close 表就可以找到一条反向的路径。

本问题中A*搜索的时间复杂度为 $O(9^9)$ 。

八数码问题

- <http://www.d.umn.edu/~jrichar4/8puz.html>
- <https://www.cs.princeton.edu/courses/archive/fall12/cos226/assignments/8puzzle.html>

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



DancingLinks 舞蹈链

- [Dancing Links - 舞蹈链](#)
 - [问题](#)
 - [重复覆盖解法:](#)
 - [精确覆盖解法:](#)
 - [源码](#)
 - [测试](#)

Dancing Links - 舞蹈链

问题

集合 $s = \{x_1, x_2, \dots, x_n\}$ 拥有 n 个成员，现在集合 s 有 m 个子集 $\{sub_1, sub_2, \dots, sub_m\}$ 。在 m 个子集中选择一些组成集合 $t = \{sub_1, sub_2, \dots\}$ ，使 t 中包含的成员可以覆盖集合 s ，即 s 中所有成员都属于 t 中的某个或某些子集。

重复覆盖：集合 s 中的任意成员 $\forall x \in t$ （允许同时属于两个以上的子集）。例如集合 $s = \{0, 1, 2, 3\}$ ，在子集 $sub_1 = \{0, 1\}$ 、 $sub_2 = \{1, 2\}$ 、 $sub_3 = \{1, 3\}$ 中选择 $t = \{sub_1, sub_2, sub_3\}$ 即可重复覆盖 s 。

精确覆盖：集合 s 中的任意成员 x 属于且只属于 t 中的一个子集，不能出现 x 不属于 t 中的任何子集，或者 x 同时属于 t 中两个以上的子集。例如集合 $s = \{0, 1, 2, 3\}$ ，在子集 $sub_1 = \{0, 1\}$ 、 $sub_2 = \{1, 2\}$ 、 $sub_3 = \{2, 3\}$ 中选择 $t = \{sub_1, sub_2\}$ 即可精确覆盖 s 。

给定集合 s 和 m 个子集，求其重复覆盖和精确覆盖。

重复覆盖解法：

遍历集合 s 中每个成员 x ，若其尚未被包含在 t 中，则在 m 个集合中寻找一个包含 x 的子集加入 t 中，重复该步骤即可获得重复覆盖。

精确覆盖解法：

对于每个元素 x_j ($1 \leq j \leq n$)，所有包含它的子集都是一种可能的选择。对于包含元素 x_j 的所有子集 $t_j = \{sub_{j-1}, sub_{j-2}, \dots, sub_{j-p}\}$ (共 p 个)，依次尝试选择每个子集 sub_{j-k} ($1 \leq k \leq p$) 作为潜在的精确覆盖。当选择子集 sub_{j-k} 时，它除了 x_j 之外还会包含其他元素，设其他元素的集合为 X_j ，那么所有其他包含 X_j 元素的子集，都与子集 sub_{j-k} 冲突，因此这次选择后，需要将 sub_{j-k}

$k\}$ 和所有与它冲突的集合都删除。

		n					
		x1	x2	...	xj	...	xn-1 xn
m	sub 1	1	0		0		1 0
	sub 2	1	0		1		0 1
	...						
	sub j-k	1	0		1		0 1
	...						
	sub m	0	0		0		0 0

上图中，当为了元素 x_j 选择子集 $sub\{j-k\}$ 后，由于 $sub\{j-k\}$ 也包含 x_1 、 x_n ，所以需要删除所有包含 x_1 、 x_n 元素的其他子集，即 sub_1 、 sub_2 。经过这次操作后，被覆盖的元素有 $\{x_1, x_j, x_n\}$ ，被选中作为精确覆盖的子集为 $sub\{j-k\}$ ，被删除的子集有 $\{sub_1, sub_2, sub\{j-k\}\}$ 。然后重复上述操作，继续考察下一个元素 x_{j+1} ，直到覆盖所有元素，则找到一组精确覆盖；若所有子集都被删除掉时，却无法覆盖到所有元素，则说明最近的一次选择是错误的，这时我们就放弃子集 $sub\{j-k\}$ ，并放弃之前的操作（被覆盖的元素中去掉 $\{x_1, x_j, x_n\}$ ，精确覆盖中去掉子集 $sub\{j-k\}$ ，被删除的子集中去掉 $\{sub_1, sub_2, sub\{j-k\}\}$ ），然后考虑下一个包含 x_j 的子集。

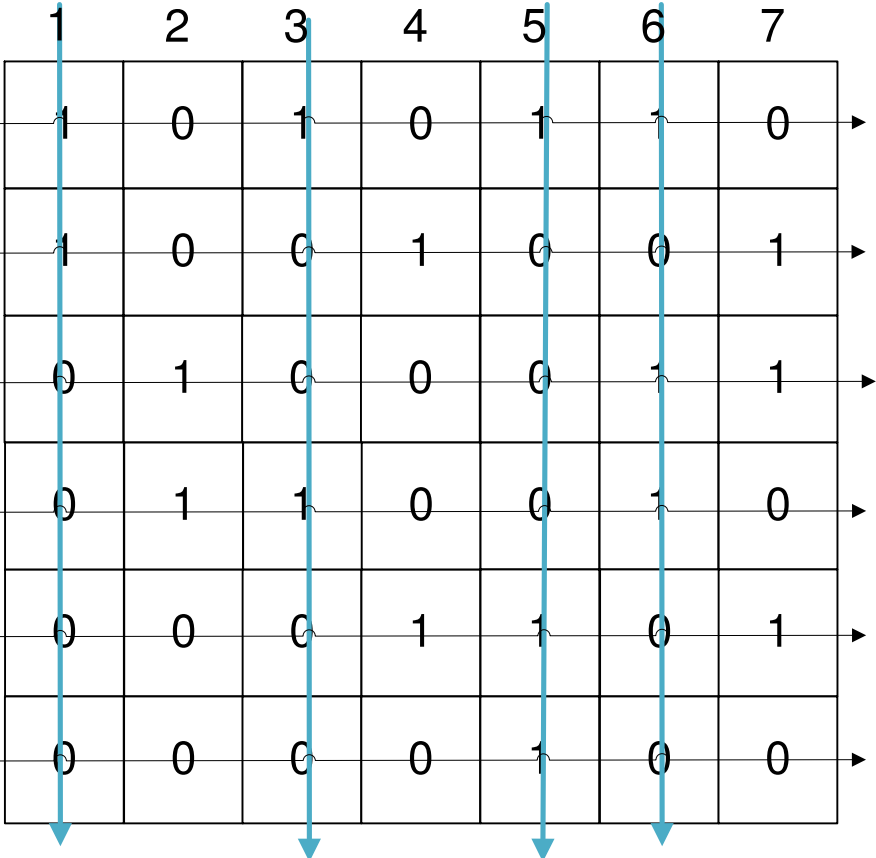
求精确覆盖的算法称为X算法，将集合 s 中的 n 个成员看作列，将 m 个子集看作行，组成一个 $m \times n$ 的矩阵 d 。若子集 sub_i （其中 $1 \leq i \leq m$ ）包含某个成员 x_j （其中 $1 \leq j \leq n$ ），则 $d[i, j] = 1$ ；若不包含则 $d[i, j] = 0$ 。

例如对于集合 $s = \{1, 2, 3, 4, 5, 6, 7\}$ ，它有 $n = 7$ 个成员，还有 $m = 6$ 个子集 $sub_1 = \{1, 3, 5, 6\}$ 、 $sub_2 = \{1, 4, 7\}$ 、 $sub_3 = \{2, 6, 7\}$ 、 $sub_4 = \{2, 3, 6\}$ 、 $sub_5 = \{4, 5, 7\}$ 、 $sub_6 = \{5\}$ 的情况，如下图所示：

		n						
		1	2	3	4	5	6	7
m	sub 1	1	0	1	0	1	1	0
	sub 2	1	0	0	1	0	0	1
	sub 3	0	1	0	0	0	1	1
	sub 4	0	1	1	0	0	1	0
	sub 5	0	0	0	1	1	0	1
	sub 6	0	0	0	0	1	0	0

(1) 从 1 开始遍历集合 s 中每个成员，对于成员 1，遍历所有子集，找到第一个满足 $d[i,1] = 1$ 的子集 sub_1 ，即 $i = 1$ 时有 $d[1,1] = 1$ ，选择该子集作为精确覆盖中的一个子集 $\{ sub_1 \}$ ，已经覆盖的成员有 $\{ 1, 3, 5, 6 \}$ ， $sub_1 = \{ 1, 3, 5, 6 \}$ 中已经包含的成员其他子集不能再出现，因此删掉其他包含 $\{ 1, 3, 5, 6 \}$ 的子集 sub_2 、 sub_4 、 sub_5 、 sub_6 、 sub_3 ，将 sub_1 也删掉；

		n						
		1	2	3	4	5	6	7
m	sub 1	1	0	1	0	1	1	0
	sub 2	1	0	0	1	0	0	1
	sub 3	0	1	0	0	0	1	1
	sub 4	0	1	1	0	0	1	0
	sub 5	0	0	0	1	1	0	1
	sub 6	0	0	0	0	1	0	0



(2) 这时矩阵 d 中所有子集都被删除，成为空矩阵，但并没有完全覆盖集合 s 中所有成员，因此 (1) 的选择是失败的，撤销 (1) 中的所有操作。继续从 1 开始遍历集合 s 中每个成员，对于成员 1，遍历所有子集，找到第二个满足 $d[i, 1] = 1$ 的子集 sub_2 ，即 $i = 2$ 时有 $d[2, 1] = 1$ ，选择该子集作为精确覆盖中的一个子集 $\{ sub_2 \}$ ，已经覆盖的成员有 $\{ 1, 4, 7 \}$ ， $sub_2 = \{ 1, 4, 7 \}$ 中已经包含的成员其他子集不能再出现，因此删掉其他包含 $\{ 1, 4, 7 \}$ 的子集 sub_1 、 sub_3 、 sub_5 ，将 sub_2 也删掉；

		n						
		1	2	3	4	5	6	7
m	sub 1	1	0	1	0	1	1	0
	sub 2	1	0	0	1	0	0	1
	sub 3	0	1	0	0	0	1	1
	sub 4	0	1	1	0	0	1	0
	sub 5	0	0	0	1	1	0	1
	sub 6	0	0	0	0	1	0	0

(3) 从 2 开始遍历集合 s 中剩下的成员，对于成员 2，遍历剩余子集，找到第一个满足 $d[i, 2] = 1$ 的子集 sub_4，即 $i = 4$ 时有 $d[4, 2] = 1$ ，选择该子集作为精确覆盖中的一个子集 $\{ \text{sub}_2, \text{sub}_4 \}$ ，已经覆盖的成员有 $\{ 1, 2, 3, 4, 6, 7 \}$ ， $\text{sub}_4 = \{ 2, 3, 6 \}$ 中已经包含的成员其他子集不能再包含，因此删掉其他包含 $\{ 2, 3, 6 \}$ 的子集（没有找到），将 sub_4 也删掉；

		n						
		1	2	3	4	5	6	7
m	sub 1	1	0	1	0	1	1	0
	sub 2	1	0	0	1	0	0	1
	sub 3	0	1	0	0	0	1	1
	sub 4	0	1	1	0	0	1	0
	sub 5	0	0	0	1	1	0	1
	sub 6	0	0	0	0	1	0	0

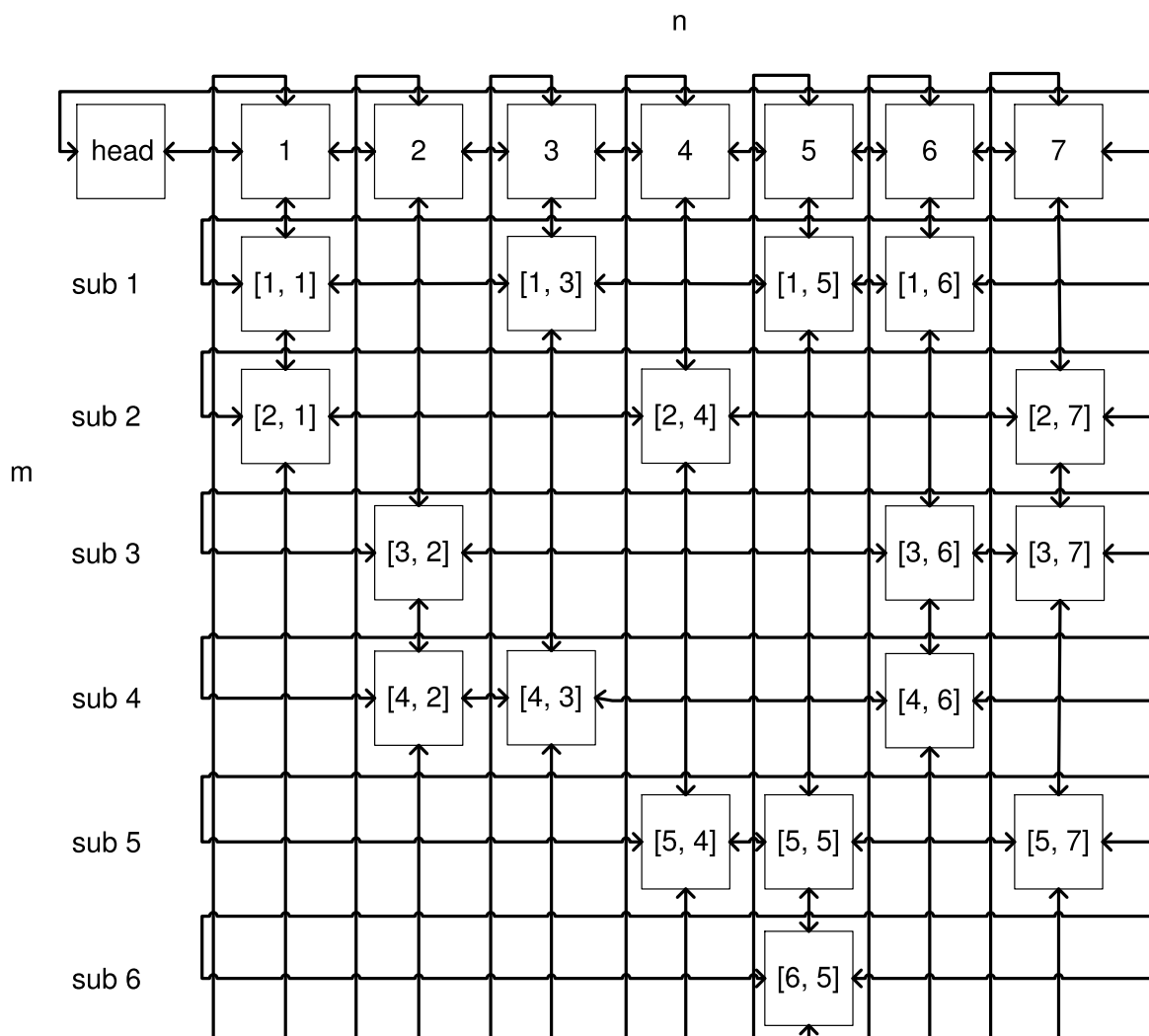
(4) 从 5 开始遍历集合 s 中剩下的成员，对于成员 5，遍历剩余子集，找到第一个满足 $d[i, 5] = 1$ 的子集 sub_6 ，即 $i = 6$ 时有 $d[6, 5] = 1$ ，选择该子集作为精确覆盖中的一个子集 $\{ sub_2, sub_4, sub_6 \}$ ，已经覆盖的成员有 $\{ 1, 2, 3, 4, 5, 6, 7 \}$ ， $sub_6 = \{ 5 \}$ 中已经包含的成员其他子集不能再包含（没有找到），将 sub_6 删掉后矩阵 d 即为空矩阵，并且已经完全覆盖了子集 s 中的所有成员，则精确覆盖的结果为 $\{ sub_2, sub_4, sub_6 \}$ ，算法结束；

		n						
		1	2	3	4	5	6	7
m	sub 1	1	0	1	0	1	1	0
	sub 2	1	0	0	1	0	0	1
	sub 3	0	1	0	0	0	1	1
	sub 4	0	1	1	0	0	1	0
	sub 5	0	0	0	1	1	0	1
	sub 6	0	0	0	0	1	0	0

(5) 当算法进行到矩阵 d 为空矩阵, 但集合 s 中所有成员并没有被完全覆盖的情况时, 说明某一次的子集选择有错误, 将该次选择的操作进行恢复, 并寻找下一个覆盖要求的子集, 继续尝试, 直到找到精确覆盖;

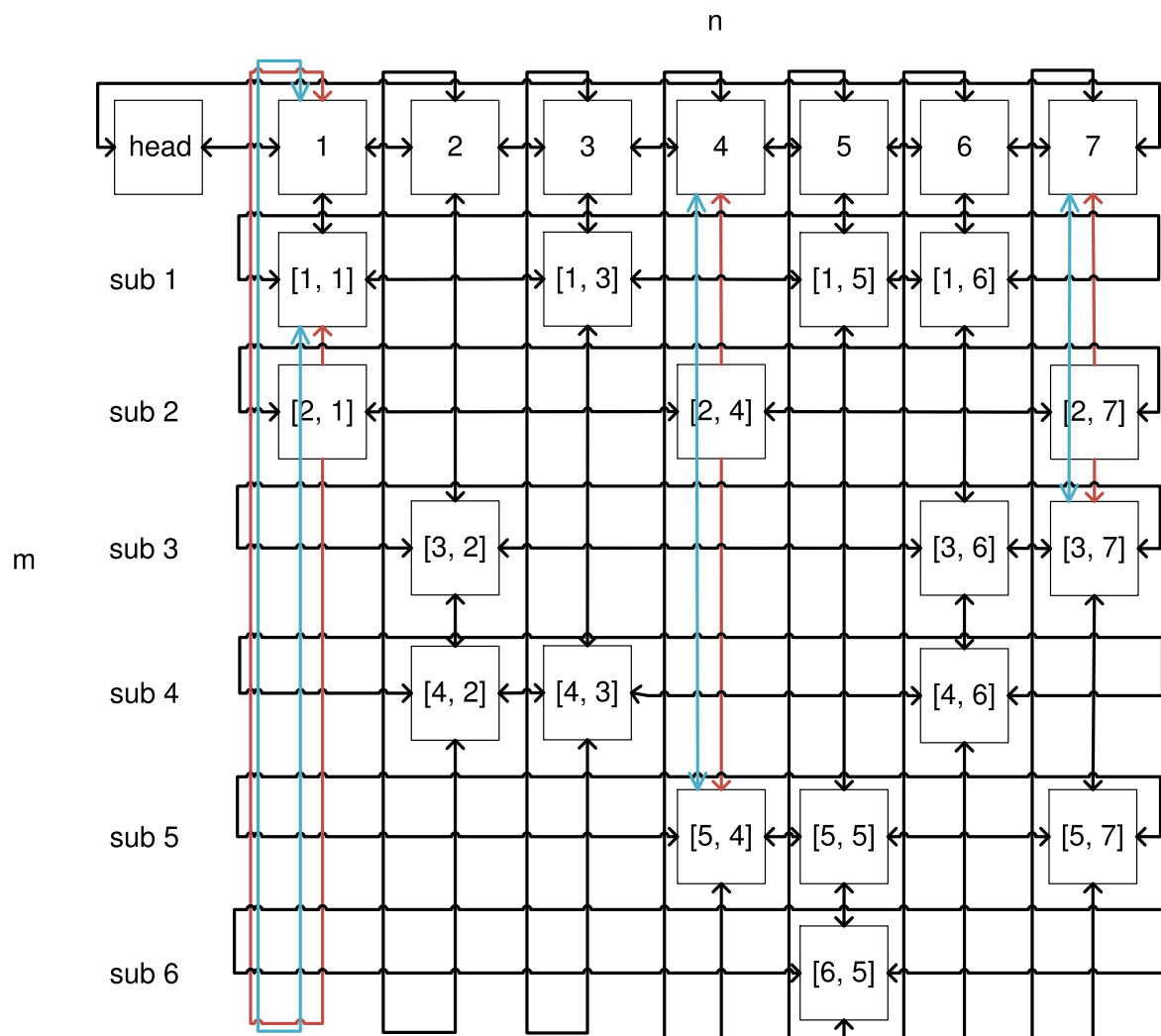
回溯法的递归结束条件是矩阵 d 为空, 每次递归时选择矩阵中的一列 x_j (其中 $0 \leq j \leq n$), 遍历矩阵 d 中的所有子集 (行), 找到一子集 sub_i (其中 $0 \leq i \leq m$) 满足 $d[i, j] = 1$, 选择该子集 (行)。由于精确覆盖的要求, 其他包含该子集中任意成员的子集, 都不能再选择, 将其删掉, 并将子集 sub_i 也删掉。重复这个操作直到将矩阵 d 删空, 检查矩阵 d 为空时是否集合 s 的所有成员都被覆盖到。在选取包含某个成员 x_j 的子集时, 可能有多个选择, 若选择其中一个子集无法最终将集合 s 完全覆盖, 则在递归函数中返回这一层, 尝试其他子集, 直到找出精确覆盖。

十字链表是一种方便删除矩阵(d)中的行列、以及恢复行列的数据结构。每个节点有上下左右(4)个指针指向周围的节点。现在将上文的集合($s = \{ 1, 2, 3, 4, 5, 6, 7 \}$)、(6)个子集以及(5)个步骤, 用十字链表的形式重复一遍。建立十字链表时需要额外对每一列添加头节点, 并添加一个总的(head)节点连接所有列的头节点, 如图所示:

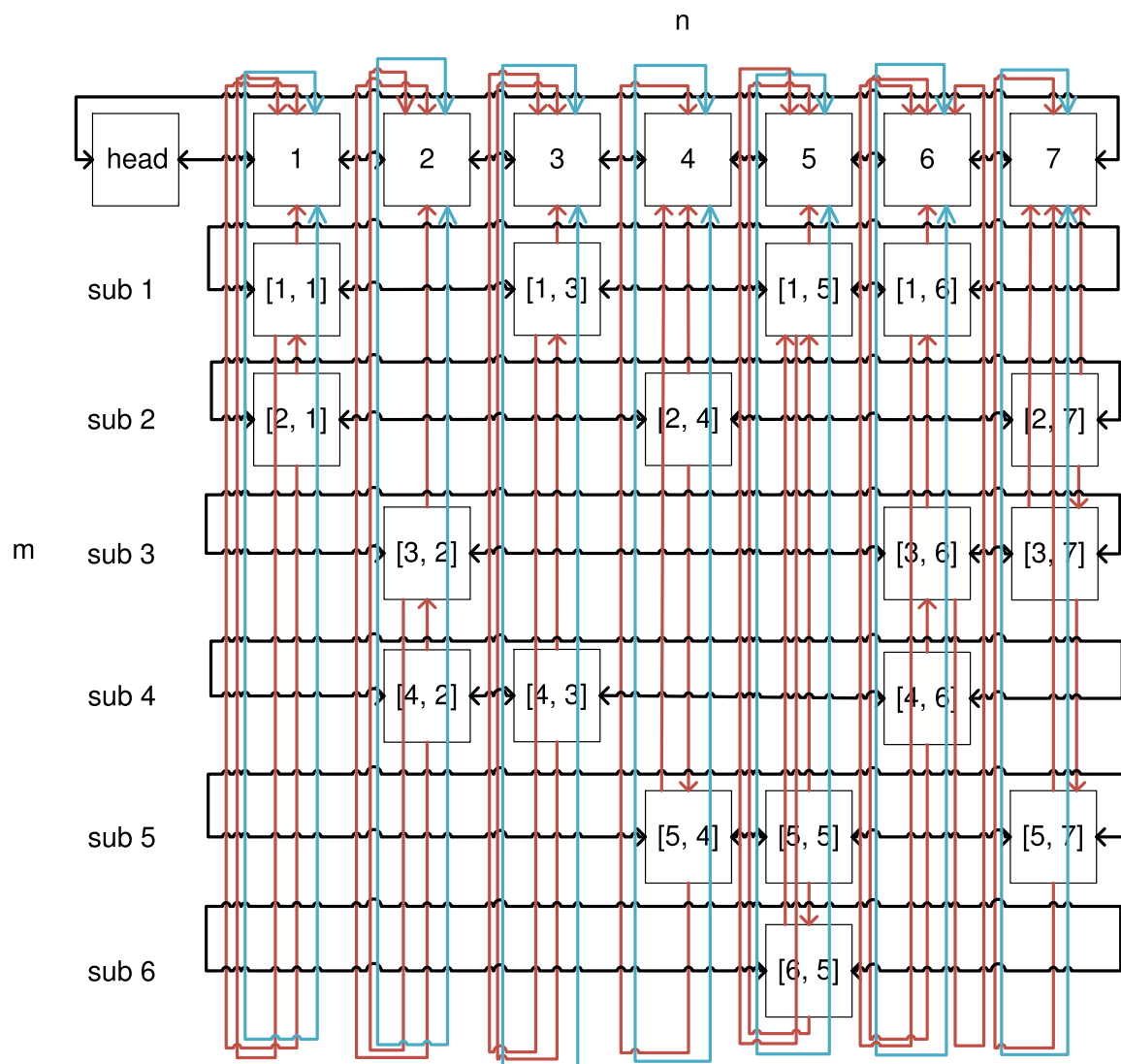


(1) 选取 head 节点右边的节点 1 (第 1 列), 在第 1 列中从上到下依次考虑每个子集, 看是否最终可以得到精确覆盖, 第 1 列有(2)个选择(sub_1)、(sub_2), 首先尝试选择(sub_1)。根据上文可知, 目标是将包含(sub_1)成员({ 1, 3, 5, 6 })的所有子集都删除掉, 即删除(sub_1)、(sub_2)、(sub_4)、(sub_5)、(sub_6)、(sub_3)。在十字链表中这个过程分为以下几个步骤来依次进行;

(2) 将 sub_1 的所有成员 [1, 1] 、 [1, 3] 、 [1, 5] 、 [1, 6] 首先删除;



(3) 再依次将属于 $\{ 1, 3, 5, 6 \}$ 列上的所有节点，以及其所在子集（行）上的所有节点，都删掉；



(4) 这时矩阵 d 为空，所选子集为 $\{ \text{sub}_1 \}$ ，覆盖的成员为 $\{ 1, 3, 5, 6 \}$ ，没有完全覆盖，因此选择错误，恢复 $\{ 1, 3, 5, 6 \}$ 列的所有元素，然后继续尝试第 1 列（节点 1）的下一个节点 $[2, 1]$ ，直到找到精确覆盖；

舞蹈链算法的时间复杂度与递归的时间复杂度一样，为 $O(n^m)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



Chapter-3 DataStructure 第3章 数据结构

- [Chapter-3 Data Structure](#)
- [第3章 数据结构](#)

Chapter-3 Data Structure

第3章 数据结构

1. [DisjointSet 并查集](#)
2. [FenwickTree\(BinaryIndexTree\) 树状数组](#)
3. [SegmentTree 线段树](#)
4. [LeftistTree\(LeftistHeap\) 左偏树 \(左偏堆\)](#)
5. [PrefixTree 前缀树](#)
6. [SuffixTree 后缀树](#)
7. [AVLTree AVL平衡树](#)
8. [RedBlackTree 红黑树](#)

DisjointSet 并查集

- [Disjoint Set - 并查集](#)
 - [描述](#)
 - [源码](#)
 - [测试](#)

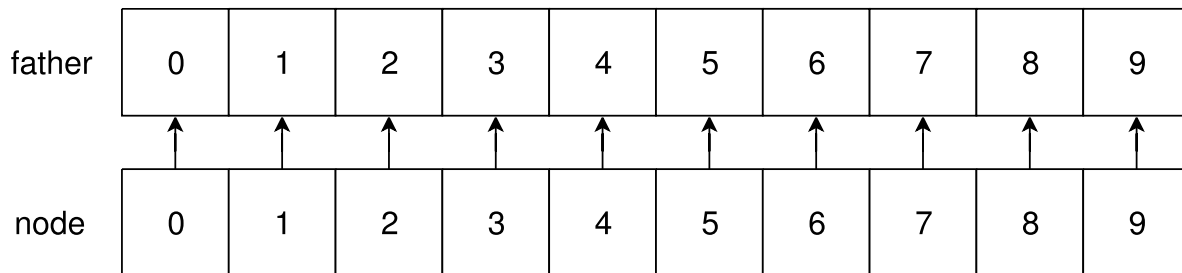
Disjoint Set - 并查集

描述

现在有一个拥有 n 个成员的集合 $s = \{ x_0, x_1, x_2, \dots, x_{n-1} \}$ ，依次声明 x_i 和 x_j 属于或不属于同一个家族，最终将所有成员分为两个家庭。每个家族中只有唯一一个祖先，其余的成员必然有一个父亲，递归的向上查找，除了祖先自己，其余每个成员所属的祖先只有 2 种可能。并查集是一种适合成员分类的高效树形数据结构，支持快速分类和查询。

设 $\text{father}[x]$ 为 x 节点的父节点，当 $\text{father}[x] = x$ 时称 x 为祖先节点，它是家族中所有其他成员共同的唯一祖先，设 $\text{ancestor}[x]$ 是 x 的祖先节点。

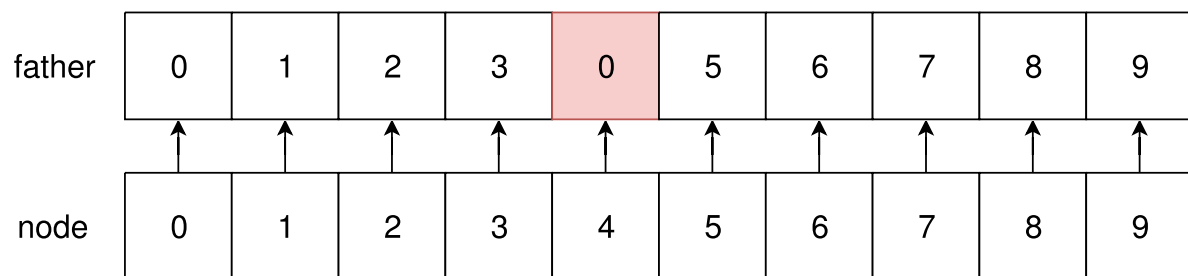
对于拥有 10 个成员的集合 $s = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$ ，将其分成两个家庭 A 和 B。初始时令每个成员的父亲都是自己，如图所示：



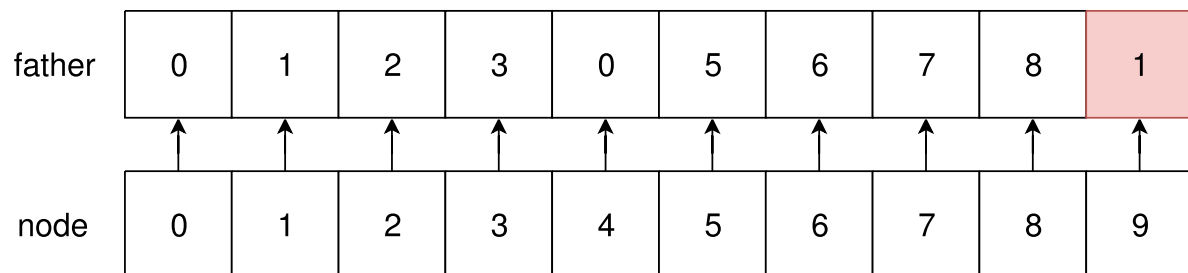
当声明 2 个成员 x_i 和 x_j ($x_i \leq x_j$) 属于同一家庭，直接令 x_i 的节点祖先为 x_j 的父亲（也可以反过来），即 $\text{father}[x_j] = \text{ancestor}[x_i]$ 。这样的操作会使元素 x_j 的最接近祖先节点，从而缩短了递归向上查找的路径长度，因此该操作也称为压缩路径。

下面对上图中的集合 s 进行具体演示：

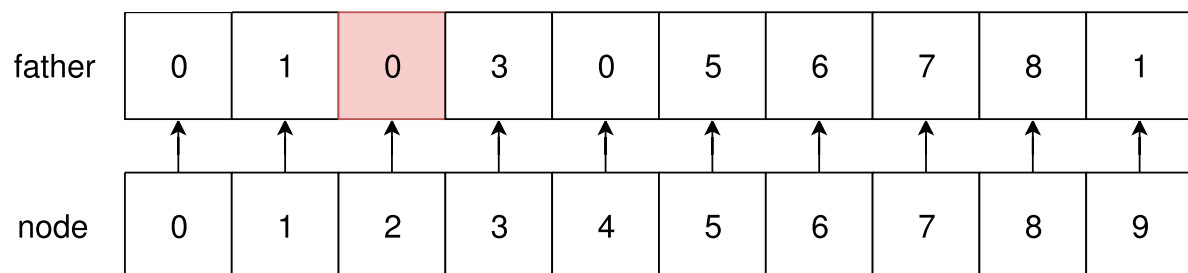
(1) 声明 0 和 4 属于同一家庭，比较 0 和 4 的祖宗节点，设置 $\text{father}[4] = \text{ancestor}[0] = 0$ ，本文中我们取左节点的祖宗节点作为右节点的父节点；



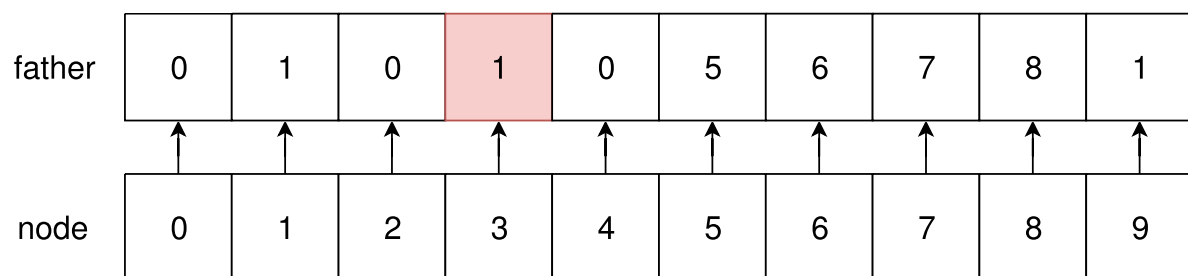
(2) 声明 1 和 9 节点属于同一家庭, 设置 $\text{father}[9] = \text{ancestor}[1] = 1$;



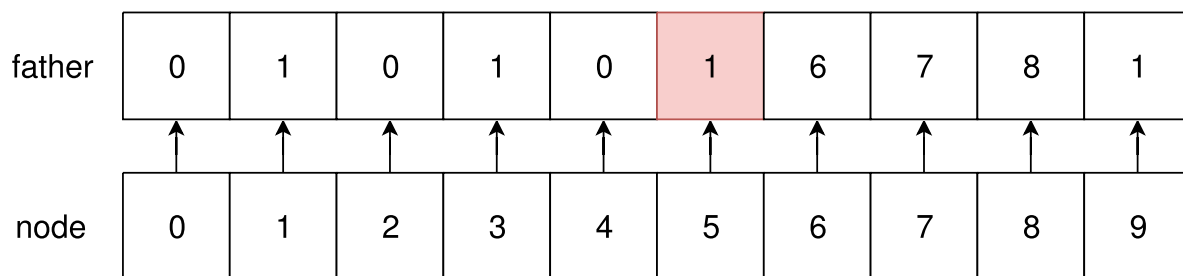
(3) 声明 0 和 2 节点属于同一家庭, 设置 $\text{father}[2] = \text{ancestor}[0] = 0$;



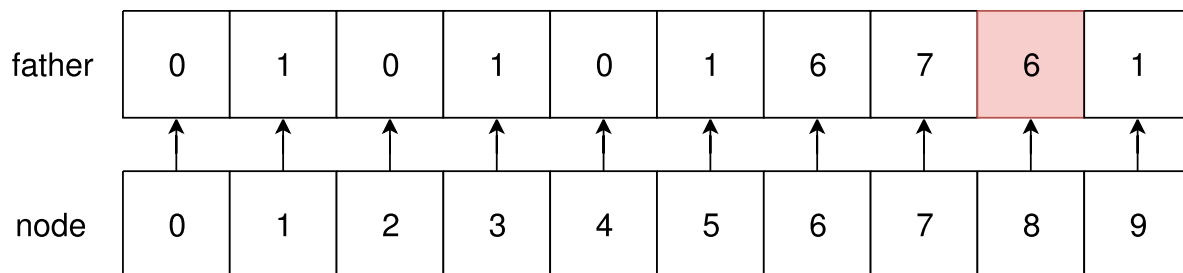
(4) 声明 1 和 3 节点属于同一家庭, 设置 $\text{father}[3] = \text{ancestor}[1] = 1$;



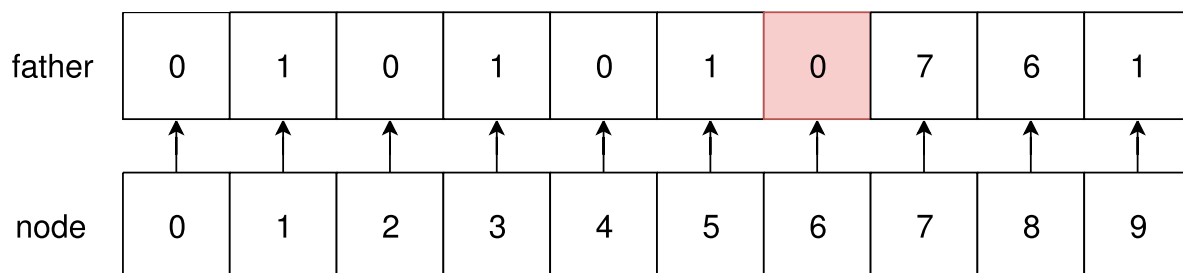
(5) 声明 3 和 5 节点属于同一家庭, 设置 $\text{father}[5] = \text{ancestor}[3] = 1$;



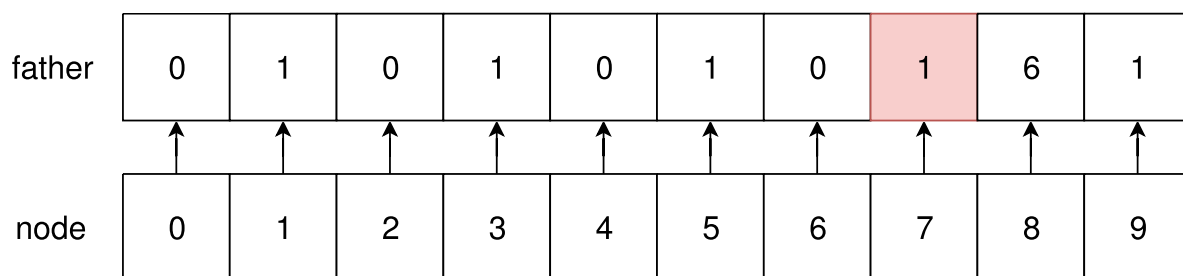
(6) 声明 6 和 8 节点属于同一家庭, 设置 $\text{father}[8] = \text{ancestor}[6] = 6$;



(7) 声明 2 和 6 节点属于同一家庭, 设置 $\text{father}[6] = \text{ancestor}[2] = 0$;



(8) 声明 1 和 7 节点属于同一家庭, 设置 $\text{father}[7] = \text{ancestor}[1] = 1$;



合并两节点 x 和 y 时, 根据固定规则设置 $\text{father}[y] = \text{ancestor}[x]$ (或者相反); 查询节点 x 的祖宗节点时, 若 $\text{father}[x] \neq \text{ancestor}[x]$ 则设置 $\text{father}[x] = \text{ancestor}[x]$ 。并查集的分类、查询操作的时间复杂度接近 $O(1)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

FenwickTree(BinaryIndexTree) 树状数组

- [Fenwick Tree\(Binary Index Tree\) - 树状数组](#)
 - [描述](#)
 - [源码](#)
 - [测试](#)

Fenwick Tree(Binary Index Tree) - 树状数组

描述

对于包含 n 个数字的数组 s ，修改其中若干成员 $s[i]$ （其中 $1 \leq i \leq n$ ）后，求数组 s 在区间 $[p, q]$ （其中 $1 \leq p \leq q \leq n$ ）上的所有成员之和。一般是在修改了成员之后，求和时遍历区间 $[p, q]$ 相加求该区间的和。修改成员 $s[i]$ （其中 $1 \leq i \leq n$ ）的时间复杂度为 $O(1)$ ，求 $s[p, q]$ 区间的和的时间复杂度为 $O(n)$ 。

而树状数组可以更快的进行区间求和，将该问题转化为前缀和，即 $s[p, q] = s[1, q] - s[1, p]$ 。类似所有整数都可以表示成 2 的幂和，也可以把一串序列表示成一系列子序列的和。其中，子序列的个数是其二进制表示中 1 的个数，并且子序列代表的 $s[i]$ 的个数也是 2 的幂。

(1) LowBit函数

函数LowBit用于计算一个数字的二进制形式下最低位的 1 代表的十进制的值。比如 $34\{10\} = 10,0010_2$ 最低位的 1 代表的十进制值为 $2\{10\}$ ， $12\{10\} = 1100_2$ 最低位的 1 代表的十进制值为 $4\{10\}$ ， $8\{10\} = 1000_2$ 最低位的 1 代表的十进制值为 $8\{10\}$ ，则有 $\text{LowBit}(34) = 2$ ， $\text{LowBit}(12) = 4$ ， $\text{LowBit}(8) = 8$ 。

在C/C++中由于补码的原因，LowBit函数实现如下：

```
1. int LowBit(int x) {
2.     return x & (x ^ (x-1));
3. }
```

或者利用计算机补码的特性，写成：

```
1. int LowBit(int x) {
2.     return x & (-x);
3. }
```

内存中的数字按照补码存储（正整数的补码与原码相同，负整数的补码是原码取反加一，并且最高位 bit 设置为 1）。比如：

$$34\{10\} = 0010,0010\{2\} \text{ , 则 } -34\{10\} = 1101,1110\{2\} \text{ ;}$$

$$12\{10\} = 0000,1100\{2\} \text{ , 则 } -12\{10\} = 1111,0100\{2\} \text{ ;}$$

$$8\{10\} = 0000,1000\{2\} \text{ , 则 } -8\{10\} = 1111,1000\{2\} \text{ ;}$$

对于非负整数 x , x 与 $-x$ 进行位与操作，即可得到 x 中最低位的 1 所代表的十进制的值。比如：

$$34\{10\} \text{ \& \& } (-34\{10\}) = 0010,0010\{2\} \text{ \& \& } 1101,1110\{2\} = 10\{2\} = 2\{10\}$$

$$12\{10\} \text{ \& \& } (-12\{10\}) = 0000,1100\{2\} \text{ \& \& } 1111,0100\{2\} = 100\{2\} = 4\{10\}$$

$$8\{10\} \text{ \& \& } (-8\{10\}) = 0000,1000\{2\} \text{ \& \& } 1111,1000\{2\} = 1000_2 = 8\{10\}$$

额外需要注意的是，CPU架构中大端模式（Big-Endian）和小端模式（Little-Endian）的区别并不会影响该计算。因为大端和小端影响的是数据在内存中存放的顺序，当数据被CPU加载到寄存器中时，所有的位操作都是在寄存器上进行的，不会影响位操作，因此位操作可以从纯数学计算的角度来看。

(2) 维护 s 前缀和的数组 bit

对于长度为 n 的数组 s （该算法需要数组下标从 1 开始，因此数组 s 的范围为 $[1,n]$ ），数组 bit 中的元素 $bit[i] = \sum_{j=i-\text{LowBit}(i)+1}^i s[j]$ 。比如：

$$\begin{array}{l} bit[1] = \sum_{j=1-1+1}^1 s[j] = s[1] \text{ \& } \\ bit[2] = \sum_{j=2-2+1}^2 s[j] = s[1]+s[2] \text{ \& } \\ bit[3] = \sum_{j=3-1+1}^3 s[j] = s[3] \text{ \& } \\ bit[4] = \sum_{j=4-4+1}^4 s[j] = s[1]+s[2]+s[3]+s[4] \text{ \& } \\ bit[5] = \sum_{j=5-1+1}^5 s[j] = s[5] \text{ \& } \\ bit[6] = \sum_{j=6-2+1}^6 s[j] = s[5]+s[6] \text{ \& } \\ bit[7] = \sum_{j=7-1+1}^7 s[j] = s[7] \text{ \& } \\ bit[8] = \sum_{j=8-8+1}^8 s[j] = s[1]+s[2]+s[3]+s[4]+s[5]+s[6]+s[7]+s[8] \text{ \& } \\ bit[9] = \sum_{j=9-9+1}^9 s[j] = s[9] \end{array}$$

在数组 `bit` 的基础上，求数组 `s` 中 `[0, p]` 的和，只需累加所有 `bit[i]`，其中初始时 `i = p`，每累加一次(`bit[i]`)，(`i`)值减去(`LowBit(i)`)，直到(`i \le 0`)。（这里我暂时也没找到更好的讲解方法，解释的不是很清晰）

对于长度为 `n` 的数组 `s`，构造树状数组的时间复杂度为 $O(n \cdot \log_2 n)$ ，查询区域和的时间复杂度为 $O(\log_2 n)$ ，修改数组 `s` 中一个值的时间复杂度为 $O(\log_2 n)$ ，空间复杂度为 $O(n)$ 。

树状数组 (Fenwick tree)：

- https://en.wikipedia.org/wiki/Fenwick_tree
 - <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.8917>
-

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

SegmentTree 线段树

- [Segment Tree - 线段树](#)
 - [描述](#)
 - [源码](#)
 - [测试](#)

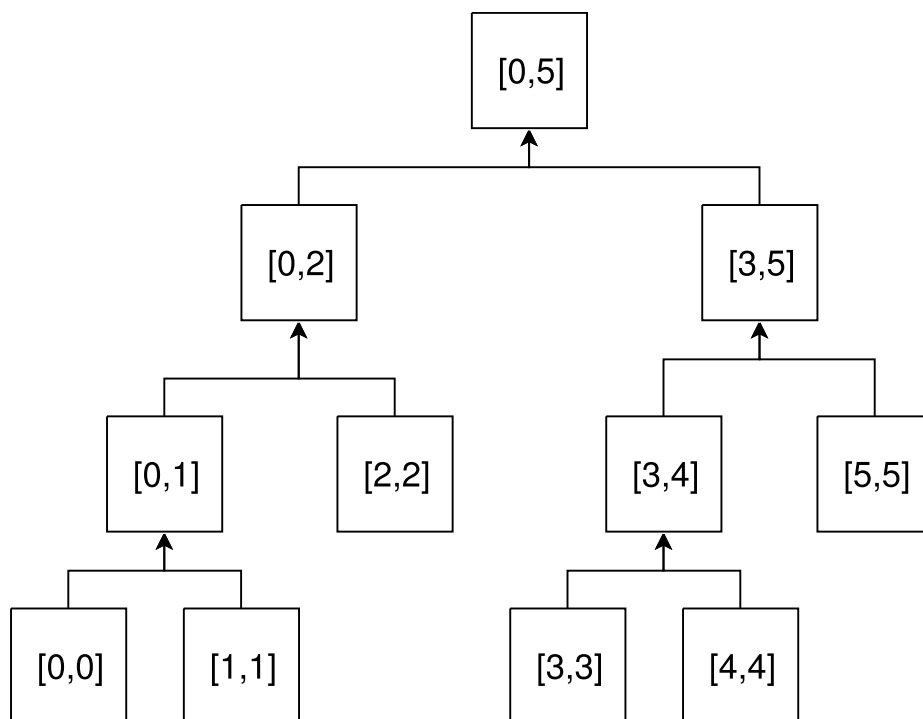
Segment Tree - 线段树

描述

线段树是一种二叉树，它将长度为 n 的数组 $s[0, n - 1]$ 划分成区间，树中的每个节点 $[i, j]$ ($0 \leq i \leq j < n$) 表示范围 $s[i, j]$ 上被关注的内容，例如该区间所有元素的和、最小元素的值、最大元素的值、第 k 大的值等。本问题关注范围 $s[i, j]$ 上所有元素之和。

在本节中我们计算该区域上所有元素的和，即节点 $[i, j]$ 代表数组 $s[i, j]$ 的和。其左子树表示 $s[i, \frac{i + j}{2}]$ 的和，右子树表示区域 $s[\frac{i + j}{2} + 1, j]$ 的和。叶子节点 $[i, i]$ (其中 $0 \leq i \leq n - 1$) 表示的区域长度为 1。

线段树 $s[0, 5]$ 如下图所示：



构造操作：从根节点开始，递归的将节点 $[i, j]$ 拆分为 $[i, \frac{i+j}{2}]$ 和 $[\frac{i+j}{2} + 1, j]$

$\frac{i+j}{2}+1, j]$ (其中 $0 \leq i \leq j < n$) , 父节点所代表的区域和等于左右孩子节点代表的区域和之和, 即 $sum[i, j] = sum[i, \frac{i+j}{2}] + sum[\frac{i+j}{2}+1, j]$, 重复该操作直到叶子节点为止。该操作的时间复杂度为 $O(n)$ 。

单点更新操作: 修改数组 s 中任意一个值 $s[i]$ (其中 $0 \leq i \leq n-1$) , 则包括该值的所有节点, 从叶子节点一直到它的所有根节点和祖先节点, 都需要修改。该操作的时间复杂度为 $O(\log_2 n)$ 。

查询操作: 从根节点向下依次查询所有子节点, 若节点属于被查询的区域则直接返回; 若节点中只有一部分区域匹配则继续查询其左右子节点。最终将所有匹配到的区域的和加起来即为查询区域的和。该操作的时间复杂度为 $O(\log_2 n)$ 。

实际编代码的时候, 对于长度为 n 的数组 $s[0, n-1]$, 为了方便我们用数组 t 来表示二叉树 (而不是真的写一个拥有两个指针的结构体) , 下标为 i 的左孩子节点下标为 $2i+1$, 右孩子节点下标为 $2i+2$ 。 $t[0]$ 为二叉树的根节点, 代表 $s[0, n-1]$ 区域的和; 其左孩子为 $t[1]$, 表示 $s[0, \frac{n}{2}]$ 区域之和; 右孩子为 $t[2]$, 表示 $s[\frac{n}{2}+1, n-1]$ 区域之和; 以此类推。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

LeftistTree(LeftistHeap) 左偏树 (左偏堆)

- [Leftist Tree\(Leftist Heap\) - 左偏树 \(左倾堆\)](#)
 - [描述](#)
 - [源码](#)
 - [测试](#)

Leftist Tree(Leftist Heap) - 左偏树 (左倾堆)

描述

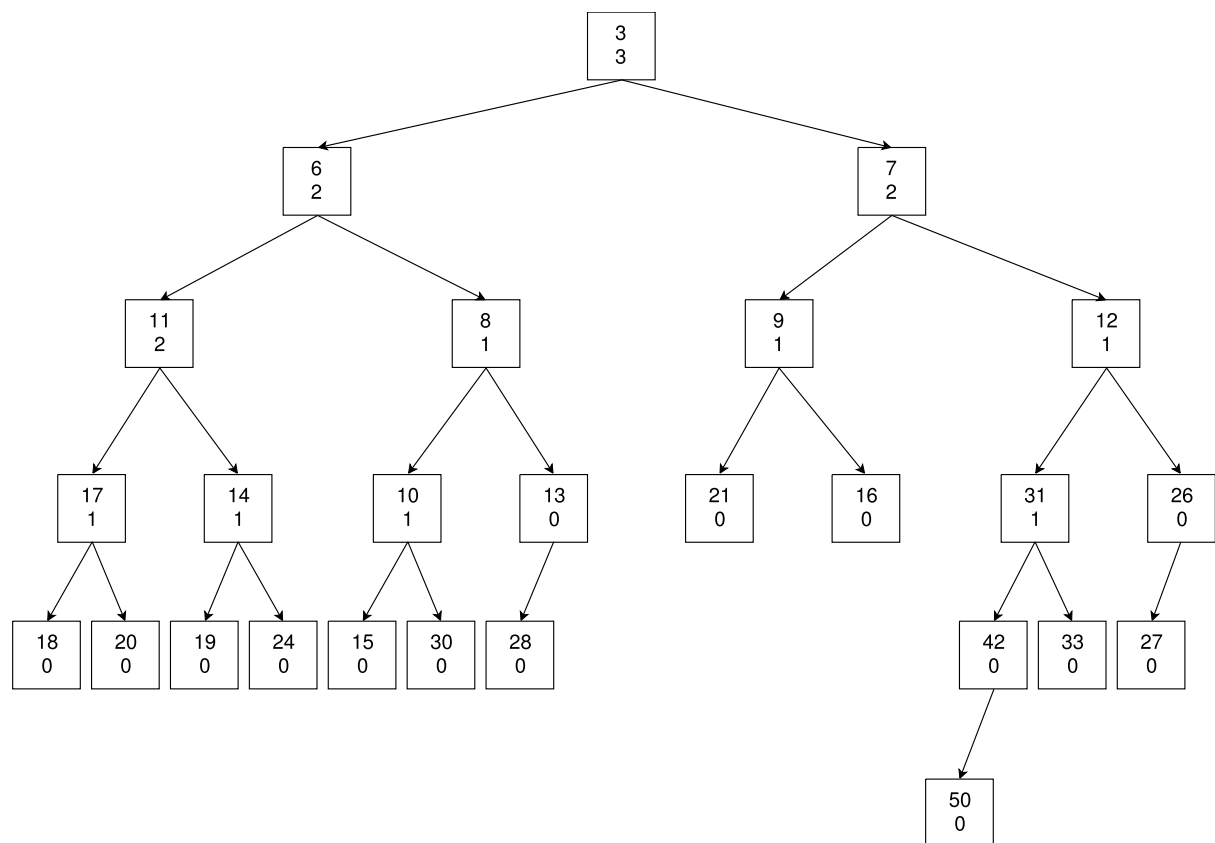
左偏树是一种接近于堆的二叉树，它的根节点总是树中的最小值或最大值。与堆不同的是，两个堆的合并需要遍历被合并的堆中所有元素，依次插入另一个堆中，而左偏树可以支持更快速的合并操作。本问题只考虑根节点为最小值情况的左偏树。

左偏树的主要操作有 (1) 合并两个左偏树； (2) 插入新节点； (3) 查找最值； (4) 删除最值。其中 (2) - (4) 的实现依赖于 (1)，合并操作是左偏树的核心。

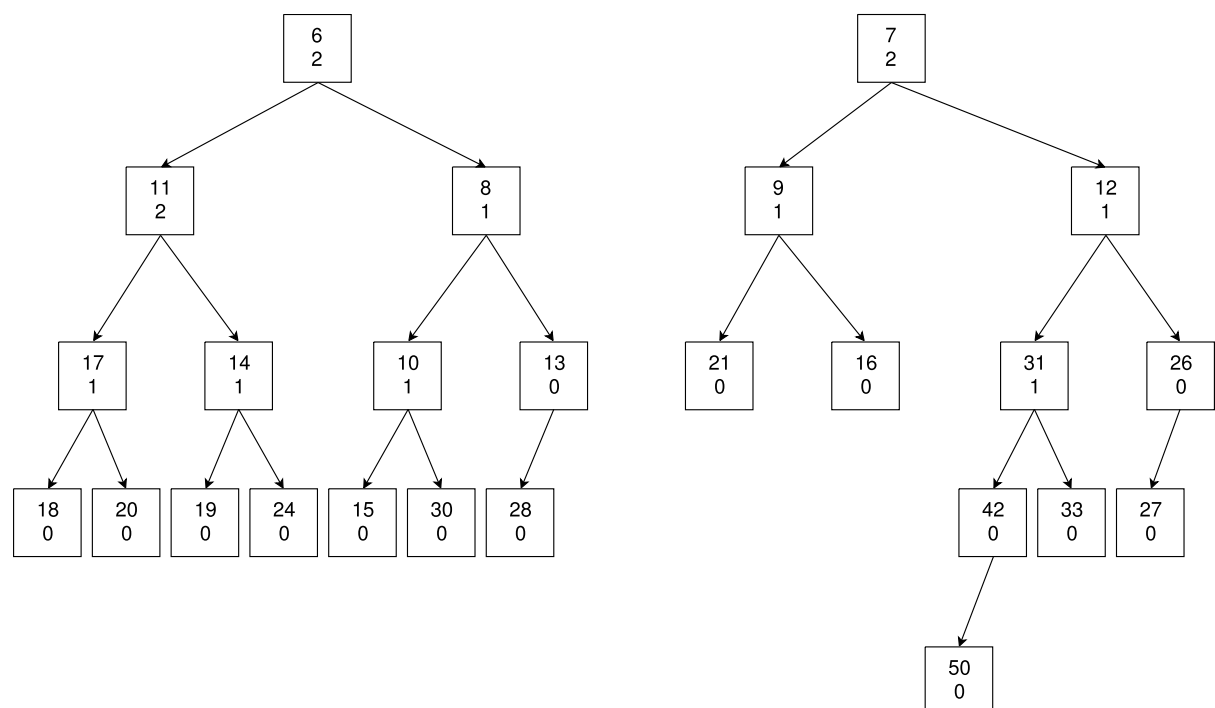
定义左偏树中一个节点的距离 d ，其值为该节点到最右下的叶子节点的距离。左偏树具有以下性质：

- (1) 父节点键值小于等于其左右孩子节点的键值， $father \leq father.leftChild$ 且 $father \leq father.rightChild$ ；
- (2) 父节点的左孩子节点的距离大于等于右孩子节点的距离， $d\{father.leftChild\} \geq d\{father.rightChild\}$ ；
- (3) 父节点的距离等于其右孩子节点的距离加 1， $d\{father\} = d\{father.rightChild\} + 1$ ；
- (4) 具有 n 个节点的左偏树的根节点的距离小于等于 $\log_2(n+1) - 1$ ， $d\{root\} \leq \log_2(n+1) - 1$ ；

下图中的每个节点中，上面的数字代表节点的下标号，下面的数字代表该节点的距离：

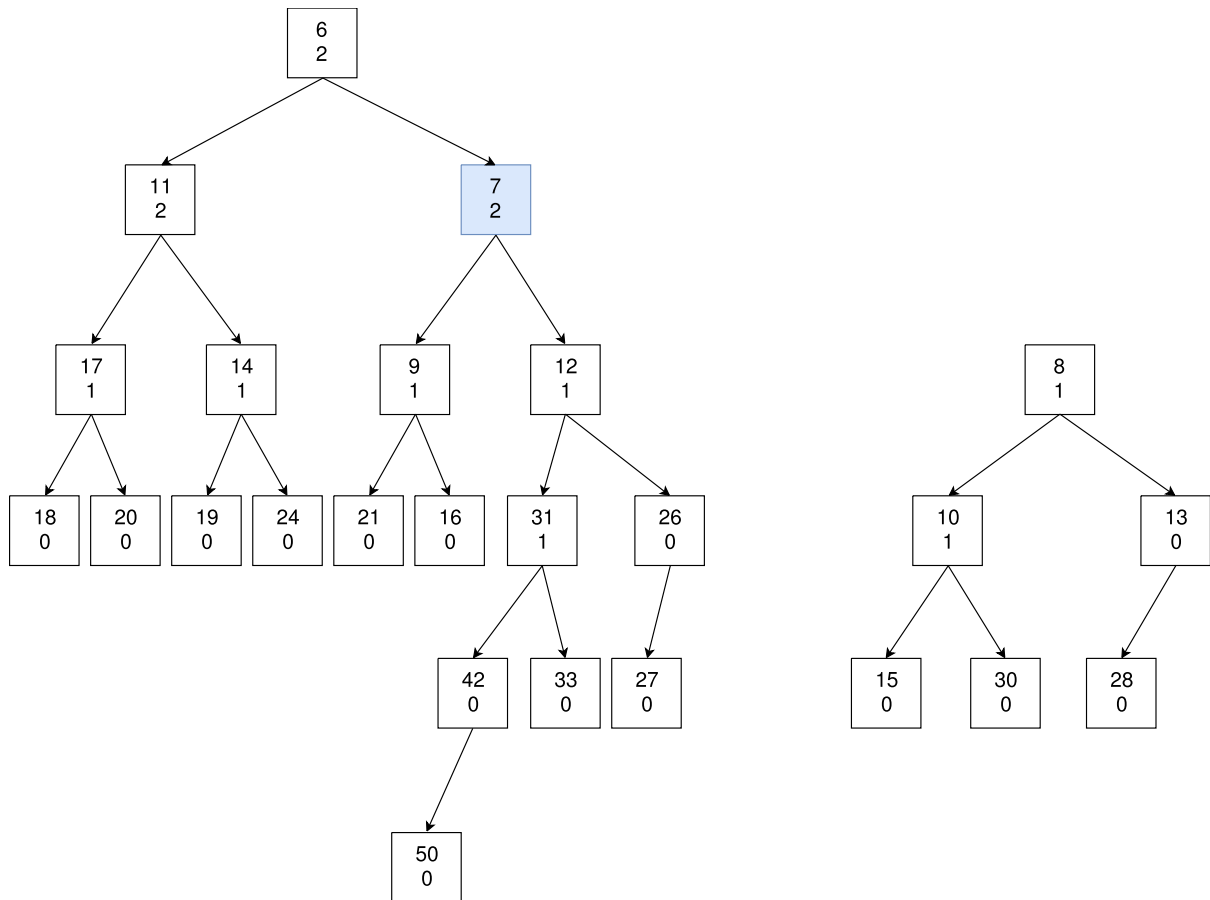


合并两个左偏树的操作，可以简单的看作（1）递归向下合并子树的根节点；（2）递归向上更新所有被修改过的节点的距离。我们通过合并下面两个左偏树来进行演示：

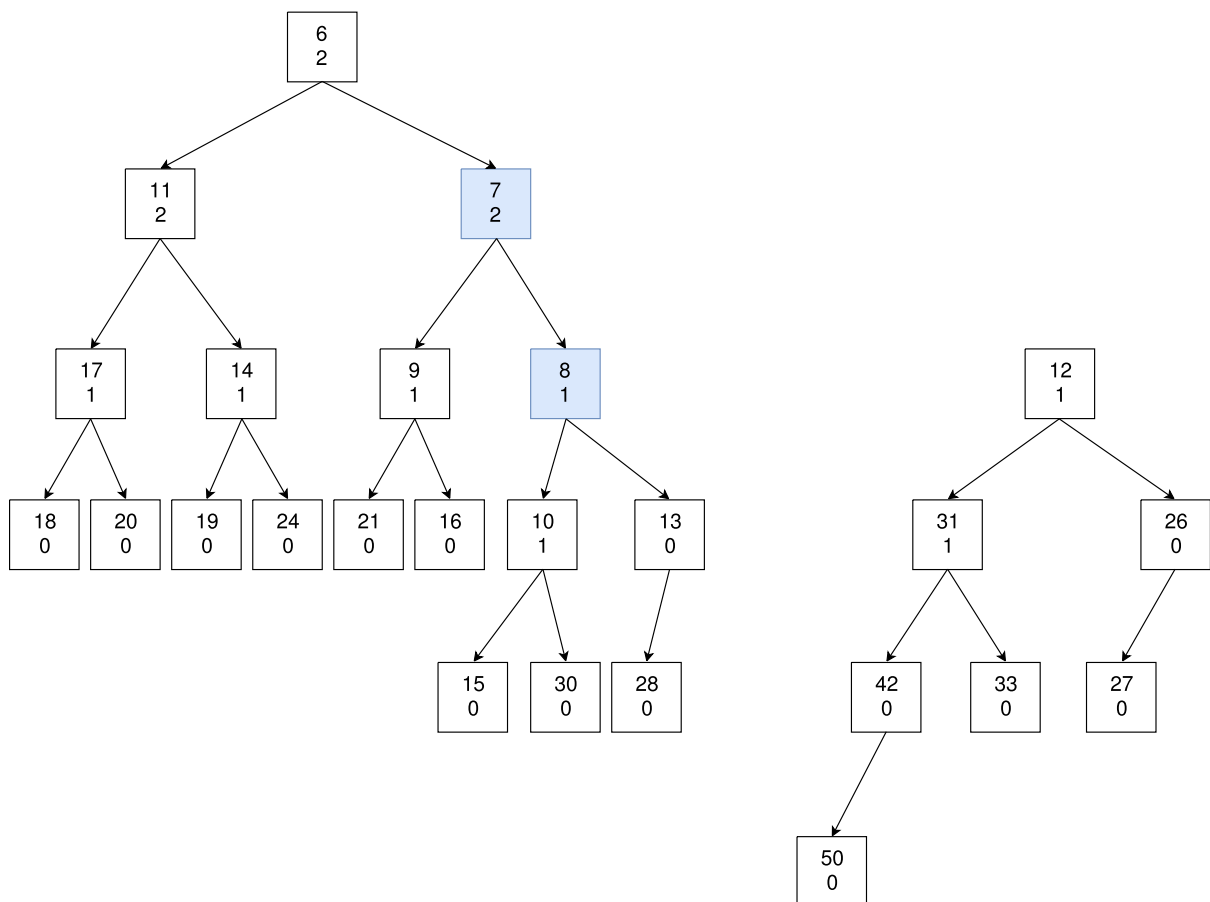


（1）比较两树根节点的值 $6 \leq 7$ ，节点 7 沿着节点 6 向右下寻找第 1 个满足 $7 \leq x$ 的节点 x ，替换 x 作为节点 6 的新右孩子节点。该节点为节点 8（ $7 \leq 8$ ）

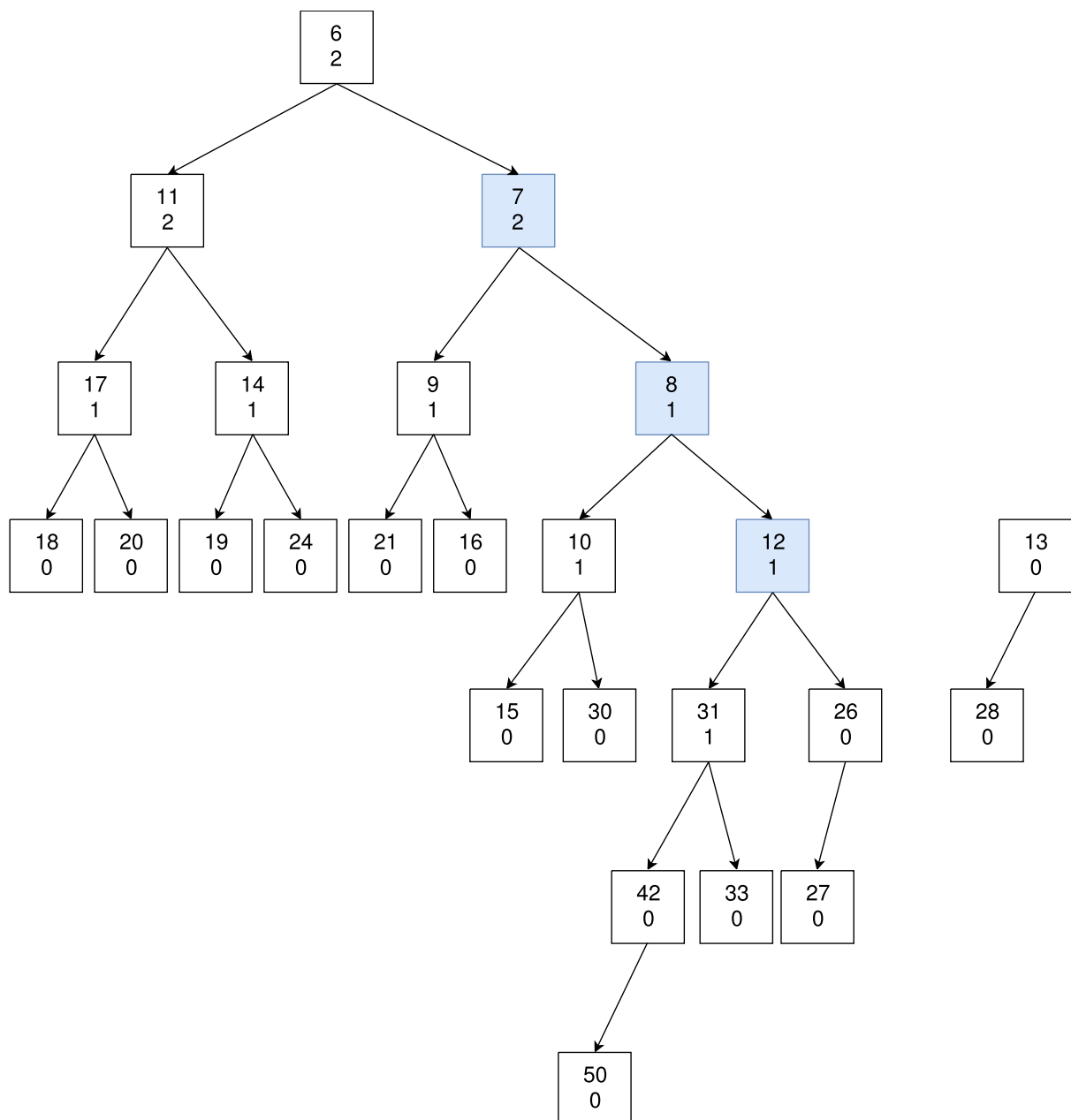
)，节点 6 的原右孩子节点 8 暂时脱离；



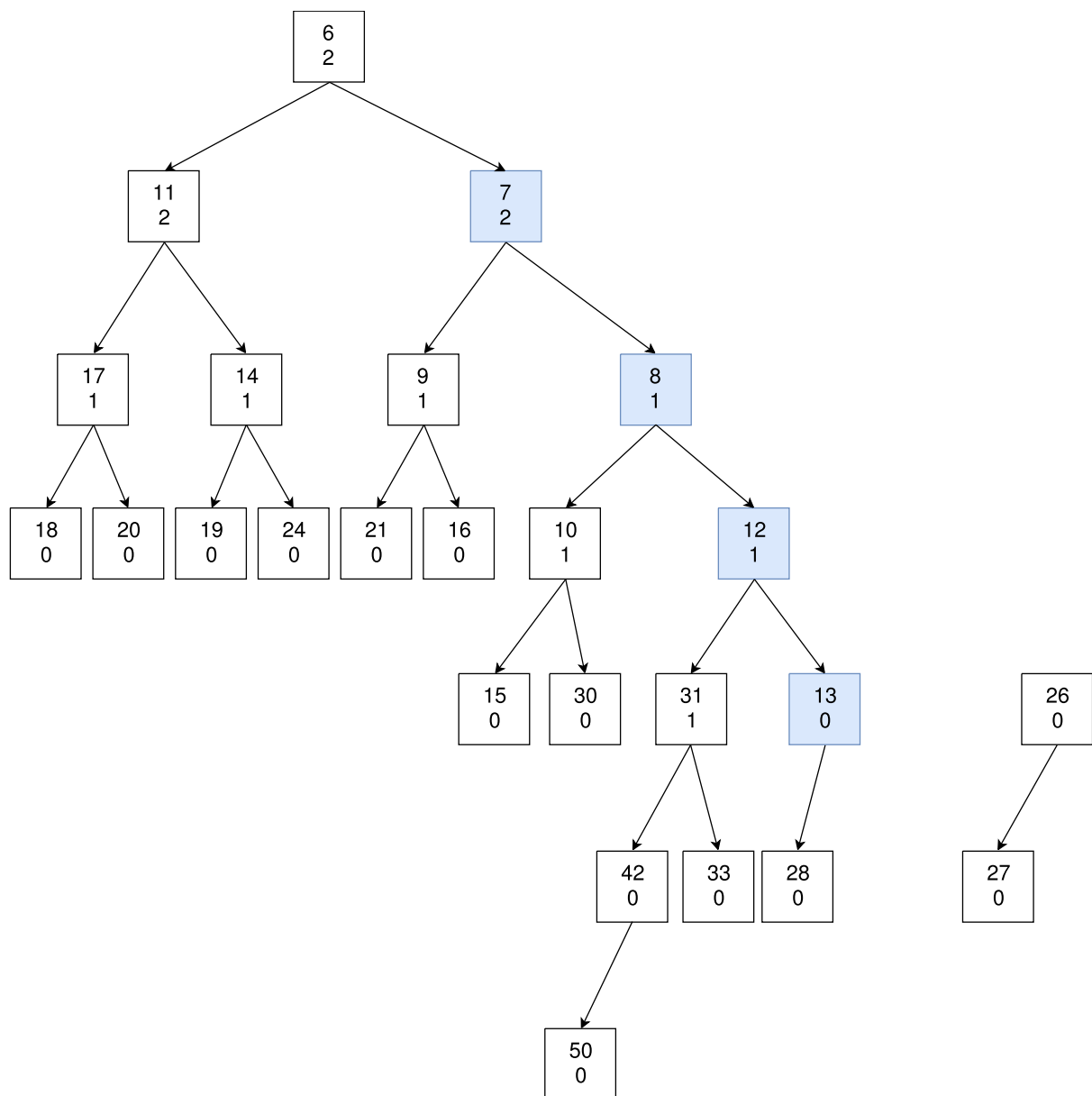
(2) 节点 8 沿着节点 7 向右下寻找第 1 个满足 $8 \leq x$ 的节点 x ，替换 x 作为节点 7 的新右孩子节点。该节点为节点 12 ($8 \leq 12$)，节点 7 的原右孩子节点 12 暂时脱离；



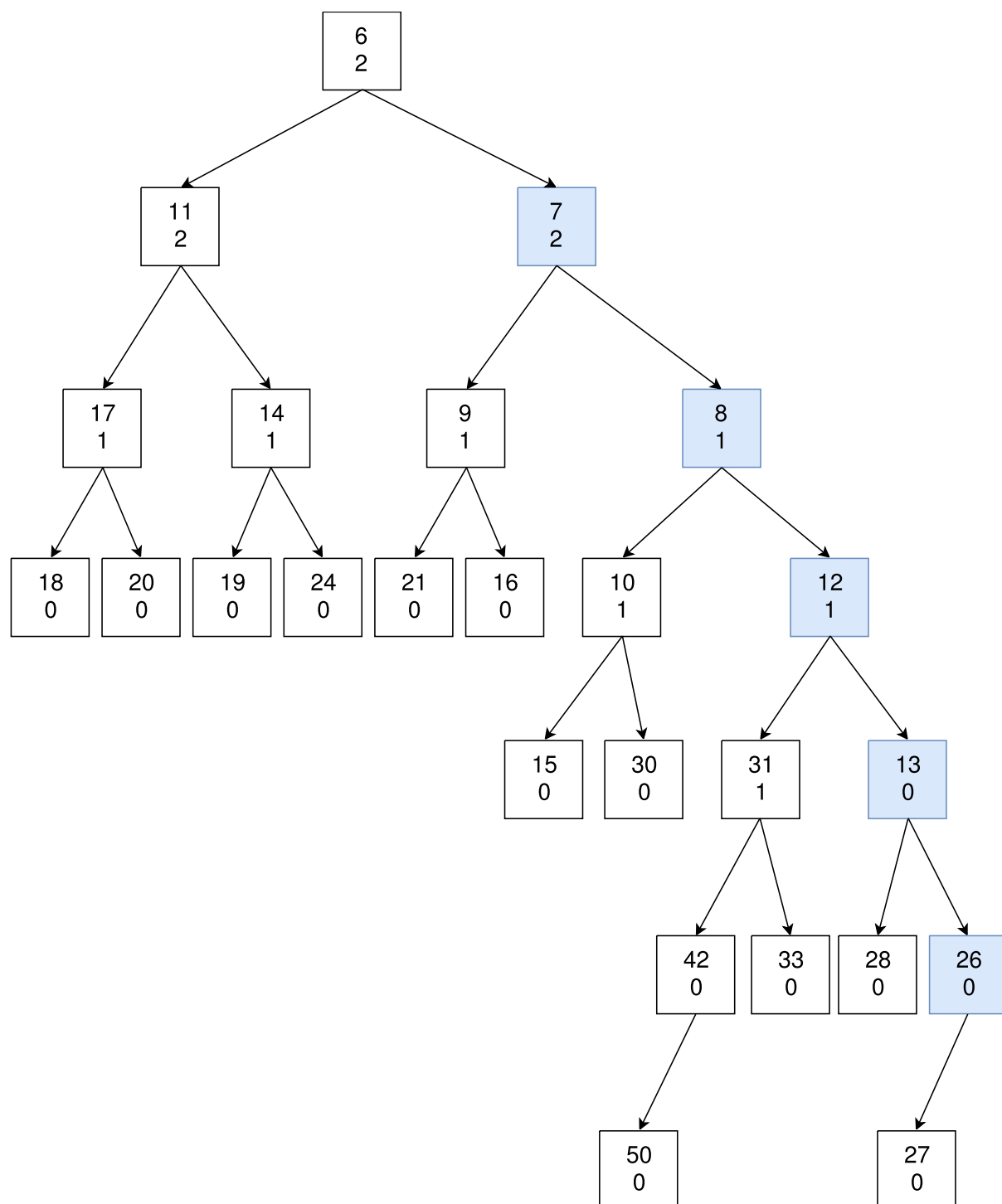
(3) 节点 12 沿着节点 8 向右下寻找第 1 个满足 $12 \leq x$ 的节点 x ，替换 x 作为节点 8 的新右孩子节点。该节点为节点 13 ($12 \leq 13$)，节点 8 的原右孩子节点 13 暂时脱离；



(4) 节点 13 沿着节点 12 向右下寻找第 1 个满足 $13 \leq x$ 的节点 x ，替换 x 作为节点 12 的新右孩子节点。该节点为节点 26 ($13 \leq 26$)，节点 12 的原右孩子节点 26 暂时脱离；



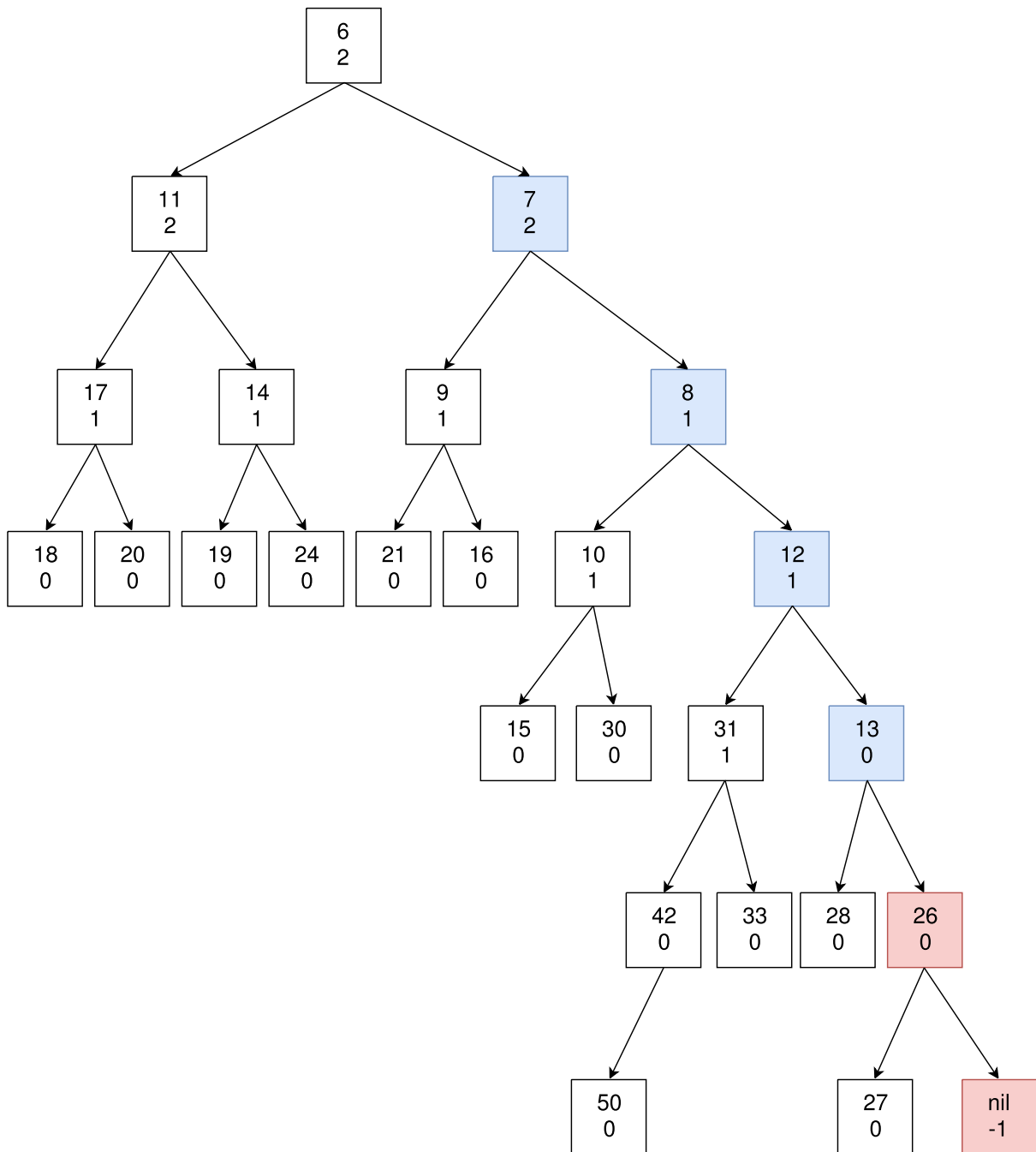
(5) 节点 26 沿着节点 13 向右下寻找第 1 个满足 $26 \leq x$ 的节点 x ，节点 13 没有右孩子节点，因此节点 26 直接成为节点 13 的右孩子节点，不再需要替换，合并操作结束；



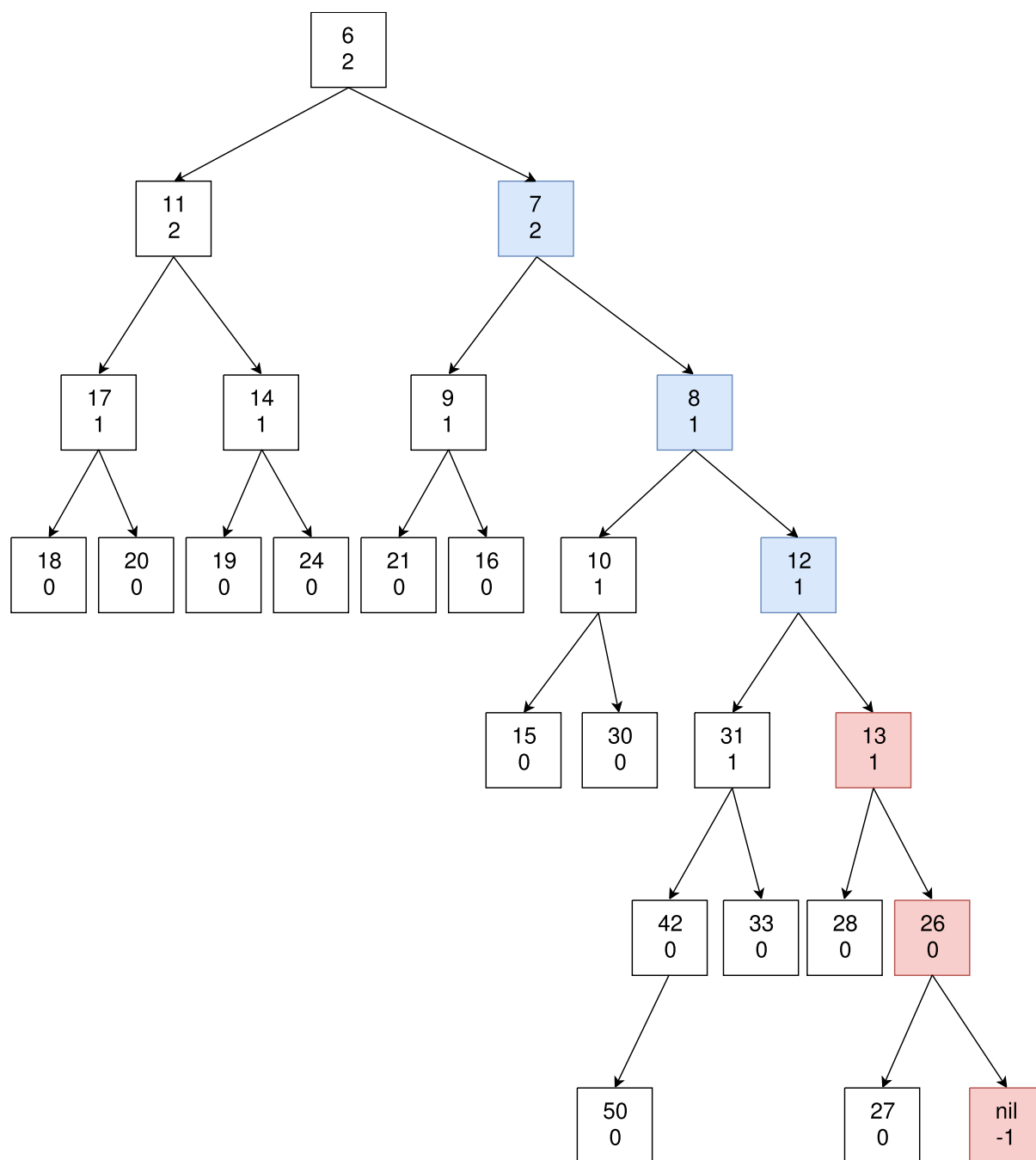
向右下插入节点的操作会影响到左偏树的平衡性，右子树变得越来越庞大。而且所有节点的距离也是错的（没有更新）。实际上每一步合并操作后还需要检查左右子树的距离属性：（1）对于 $d_{\text{leftChild}} < d_{\text{rightChild}}$ 的情况，交换左右子树；（2）对于 $d_{\text{root}} \neq d_{\text{rightChild}} + 1$ 的情况，更新 d_{root} 。

节点距离的更新必须从叶子节点开始向上进行，对于之前步骤中修改过的节点，重新遍历计算其距离（其中空节点的距离认为 -1 ）：

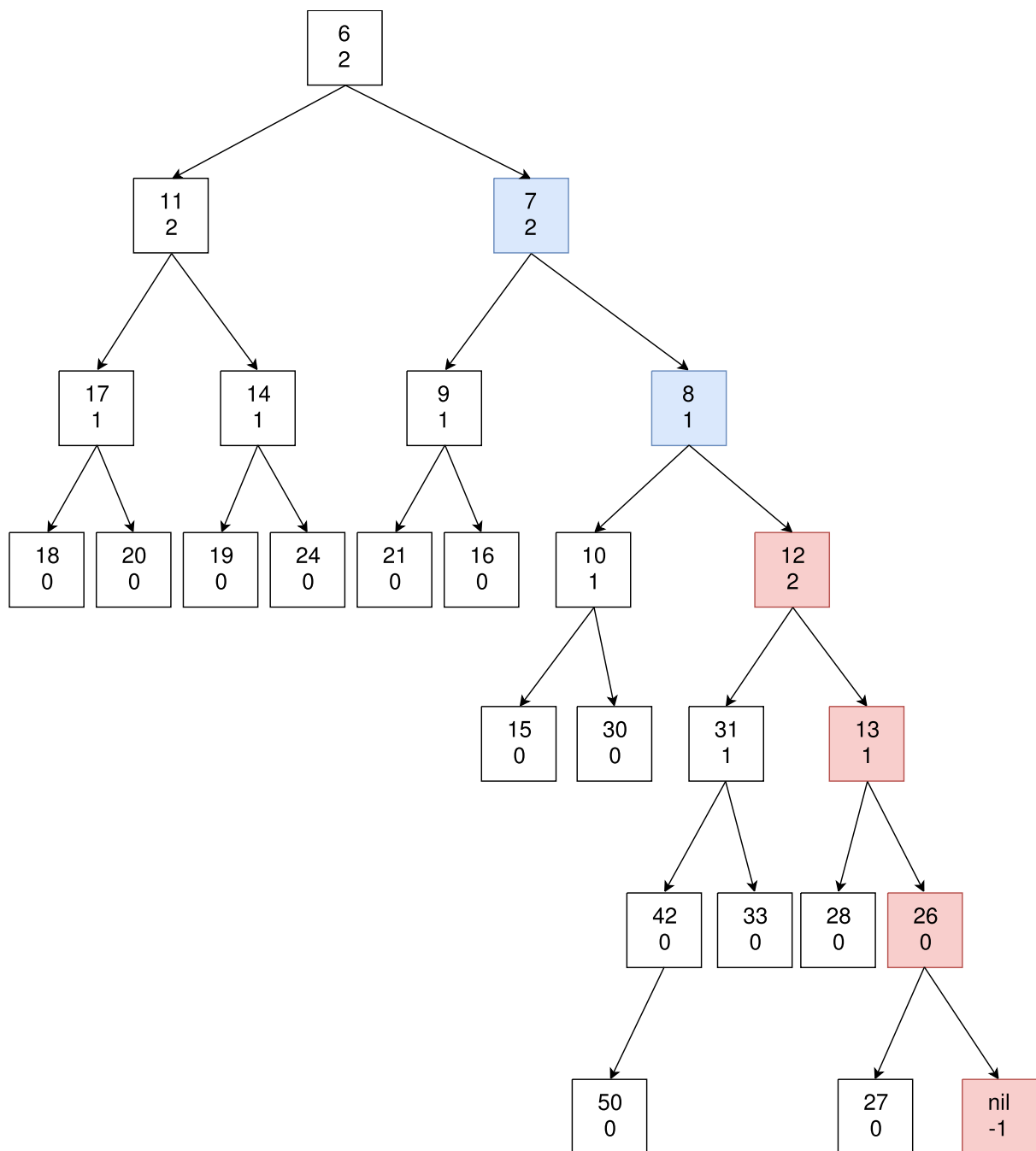
(6) 对于节点 26 , $d\{26.leftChild\} = d\{27\} = 0 \geq d\{26.rightChild\} = \{d\{nil\}\} = -1$, 不需要交换左右子树, $d\{26\} = d\{26.rightChild\} + 1 = d\{nil\} + 1 = -1 + 1 = 0$;



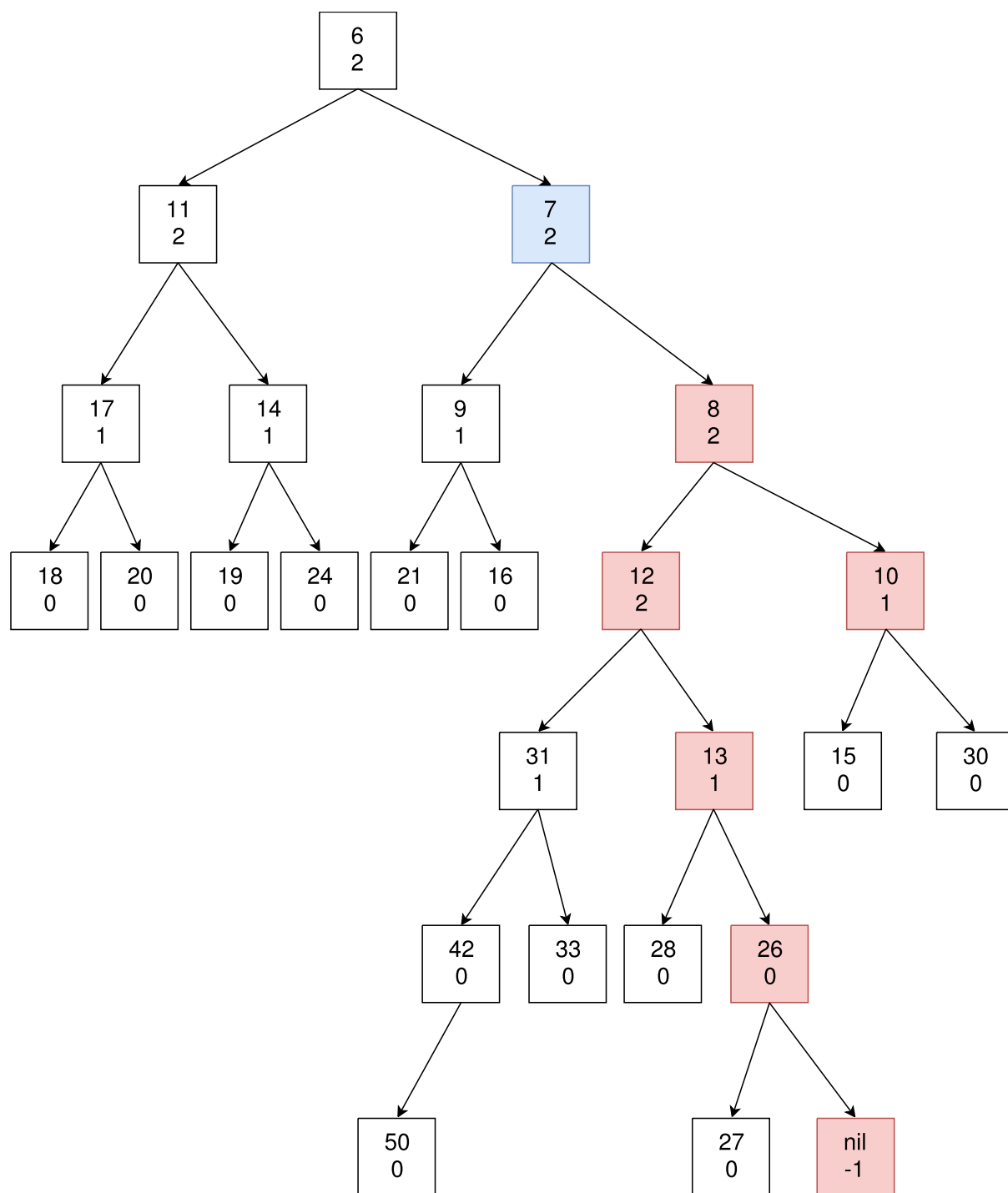
(7) 对于节点 13 , $d\{13.leftChild\} = d\{28\} = 0 \geq d\{13.rightChild\} = d\{26\} = 0$, 不需要交换左右子树, $d\{13\} = d\{13.rightChild\} + 1 = d\{26\} + 1 = 0 + 1 = 1$;



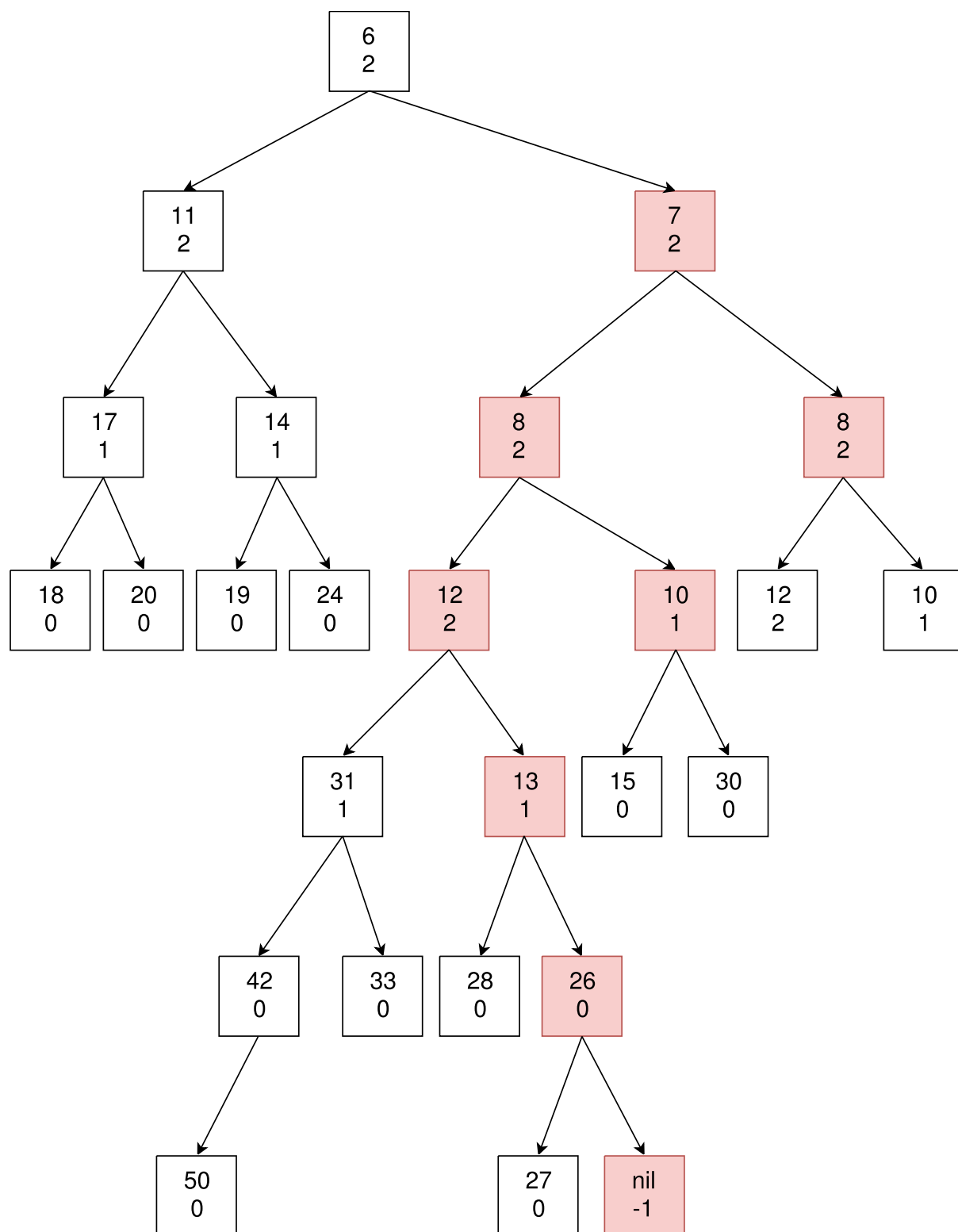
(8) 对于节点 12 , $d\{12.leftChild\} = d\{31\} = 1 \geq d\{13.rightChild\} = d\{26\} = 1$, 不需要交换左右子树, $d\{12\} = d\{12.rightChild\} + 1 = d\{13\} + 1 = 1 + 1 = 2$;



(9) 对于节点 8 , $d\{8.leftChild\} = d\{10\} = 1 \wedge d\{8.rightChild\} = d\{12\} = 2$, 需要交换子树 10 和子树 12 , $d\{8\} = d\{8.rightChild\} + 1 = d\{10\} + 1 = 1 + 1 = 2$;



(10) 对于节点 7 , $d\{7.leftChild\} = d\{9\} = 1$ \wedge $d\{7.rightChild\} = d\{8\} = 2$, 需要交换子树 9 和子树 8 , $d\{7\} = d\{7.rightChild\} + 1 = d\{9\} + 1 = 1 + 1 = 2$;



(11) 对于节点 6 , $d\{6.leftChild\} = d\{11\} = 2 \geq d\{6.rightChild\} = d\{7\} = 2$, 不需要交换左右子树, $d\{6\} = d\{6.rightChild\} + 1 = d\{7\} + 1 = 2 + 1 = 3$;

实际编码时可以通过递归的方式将合并子树和更新距离两个操作放在同一个函数中, 合并函数Merge返回合并后左偏树的根节点的距离 d , 在Merge中调用左右孩子的合并操作, 获取左右孩子的距

离，然后再决定是否交换左右子树，并更新父节点的距离。本文的将两个步骤分开是为了方便理解算法。

左偏树的插入操作，可以看作左偏树与一个只有根节点的左偏树的合并操作；删除最值的操作，可以看作删除根节点后，合并左右子树的操作。

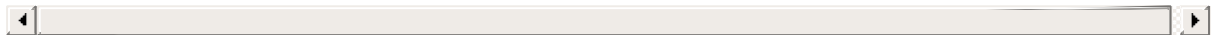
左偏树的合并操作、插入节点操作、删除根节点操作的时间复杂度都为 $O(\log_2 n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



PrefixTree 前缀树

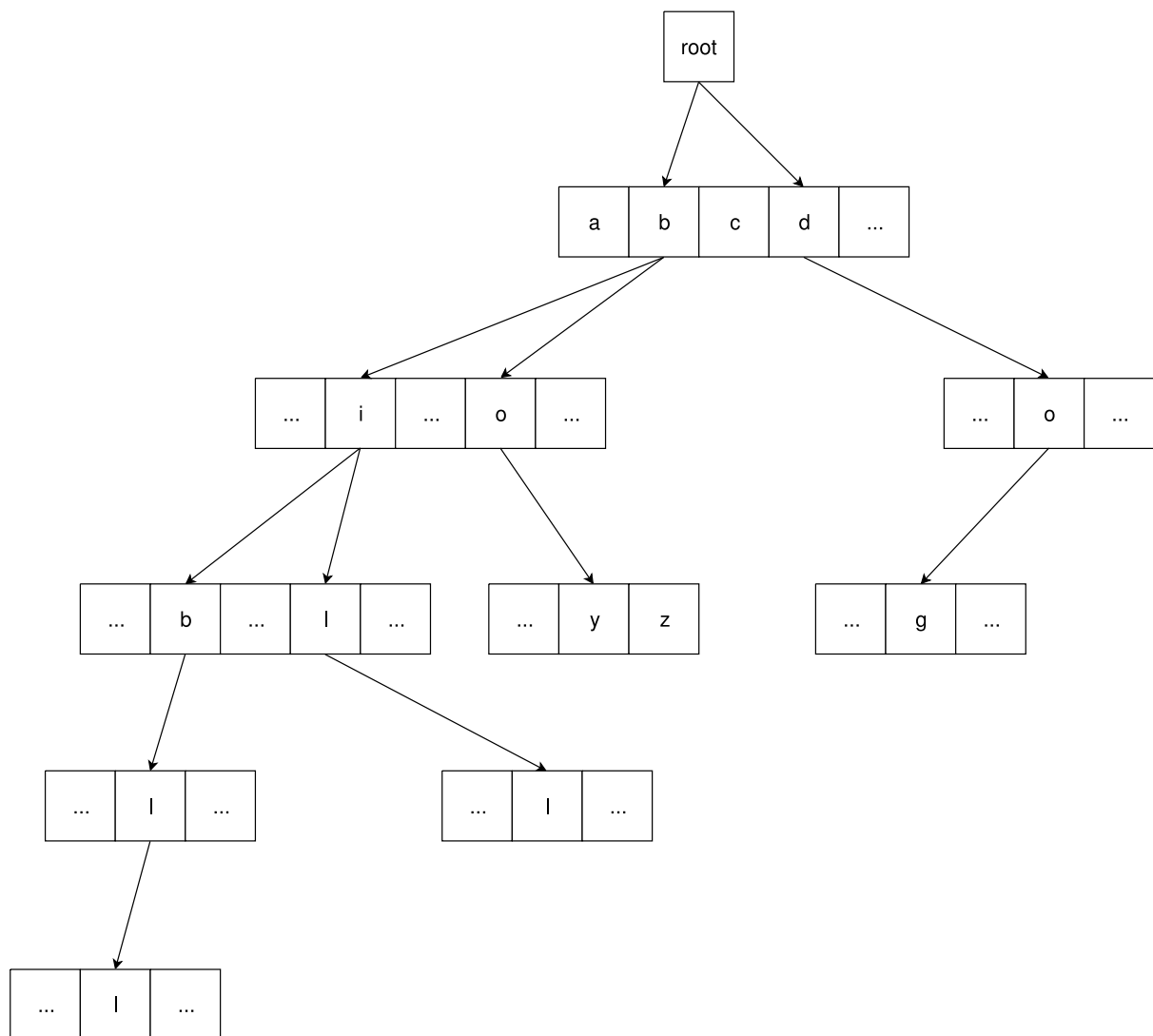
- [Prefix Tree - 前缀树](#)
 - [描述](#)
 - [源码](#)
 - [测试](#)

Prefix Tree - 前缀树

描述

前缀树是一种支持在一个英文单词集合中快速查询是否包含某个单词，或者包含某个前缀的一些单词的数据结构，或者记录某个单词出现过的次数。

为了简化问题，本问题假设字典中只会出现 a - z 这 26 个小写字母，前缀树中每个节点都包含 26 个孩子节点，将跟节点的孩子节点作为所有单词的首字母，向下查找直到跟节点就可以得到完整的单词。一个包含 boy 、 dog 、 bible 、 bill 的前缀树如图：



这样每次查找单词时，按照前缀从根节点开始向下匹配每个孩子节点的字符即可。前缀树查找一个长度为 n 的单词的时间复杂度为 $O(n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```


SuffixTree 后缀树

- [Suffix Tree - 后缀树](#)
 - [描述](#)
 - [源码](#)
 - [测试](#)

Suffix Tree - 后缀树

描述

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

AVLTree AVL平衡树

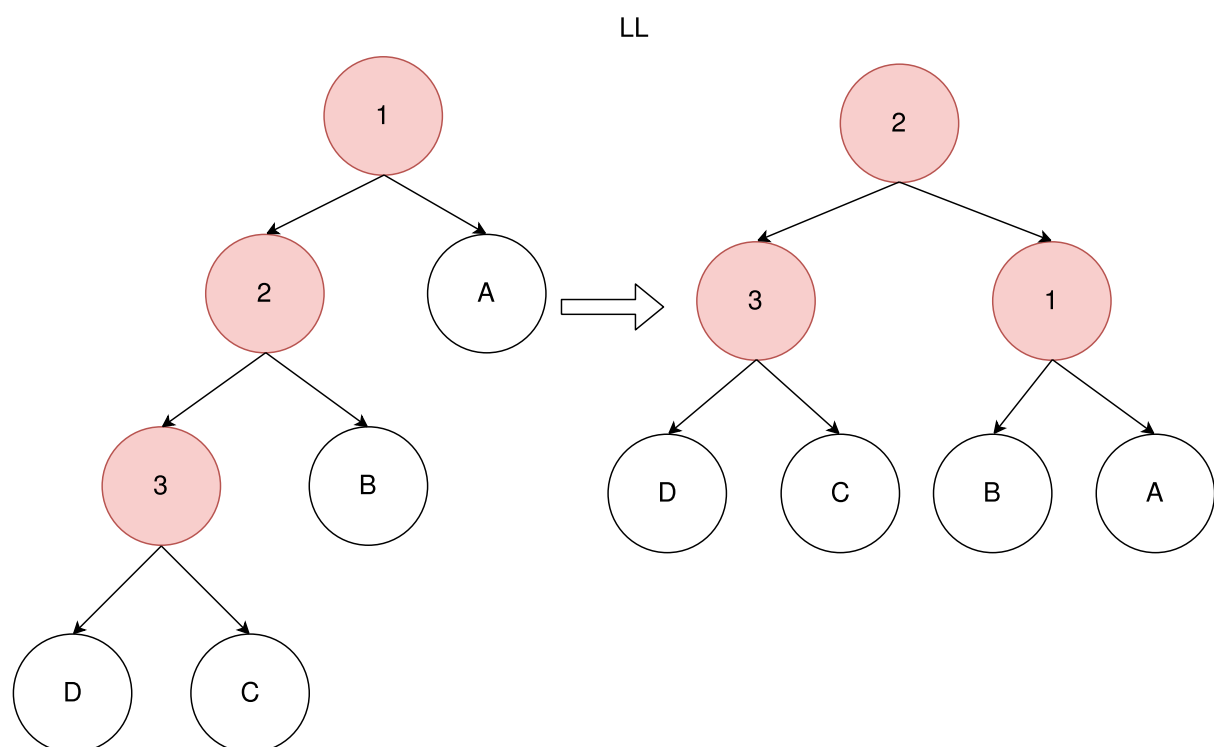
- [AVL Tree - AVL二叉树](#)
 - [描述](#)
 - [源码](#)
 - [测试](#)

AVL Tree - AVL二叉树

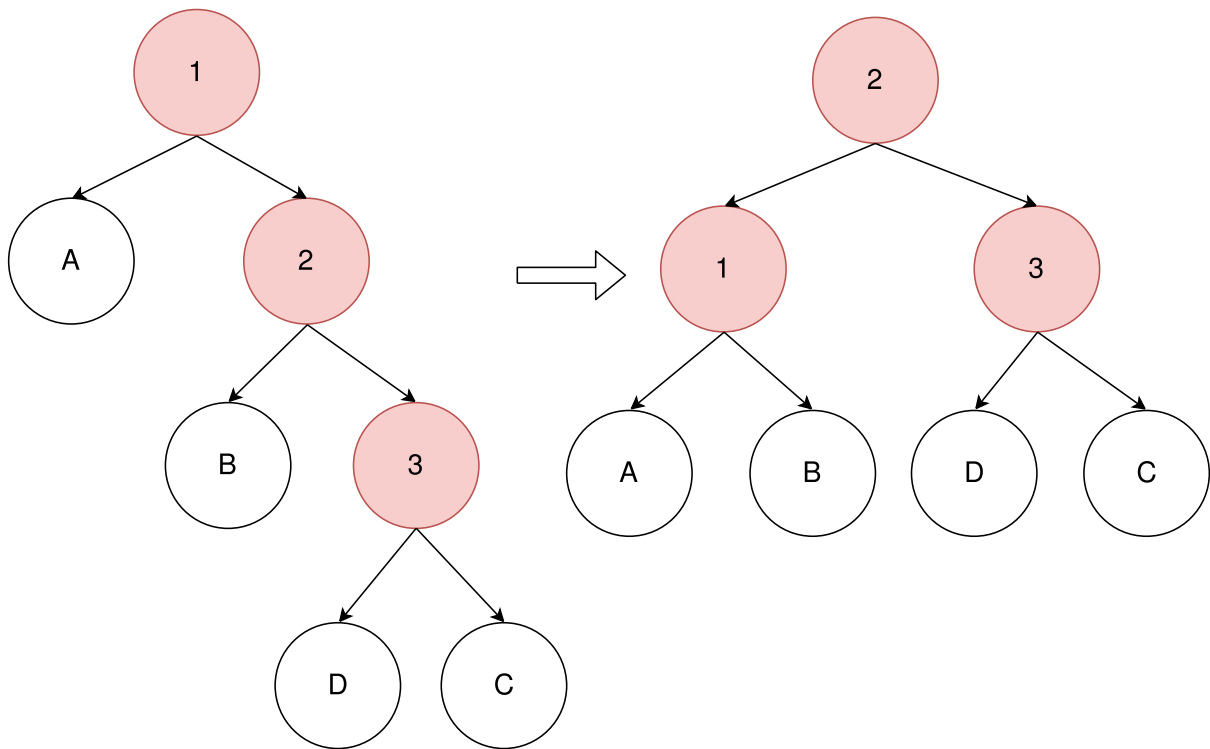
描述

AVL树是最早发明的一种自平衡二叉查找树，树中的任何节点的左右两个子树的高度最大差别为 1，因此也称为高度平衡树。AVL树的查找、插入、删除操作的平均时间复杂度都是 $O(\log_2 n)$ ，AVL树高度为 $O(\log_2 n)$ 。

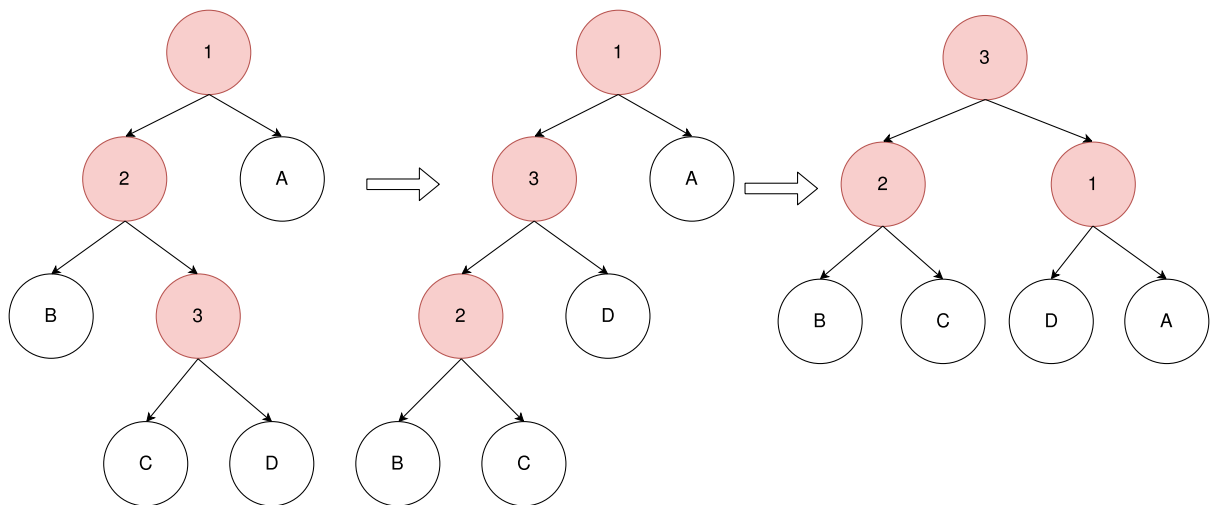
为了保持树的左右子树的平衡，避免一侧过长或过短，AVL树会对LL（左左）、RR（右右）、LR（左右）、RL（右左）四种情况进行调整：

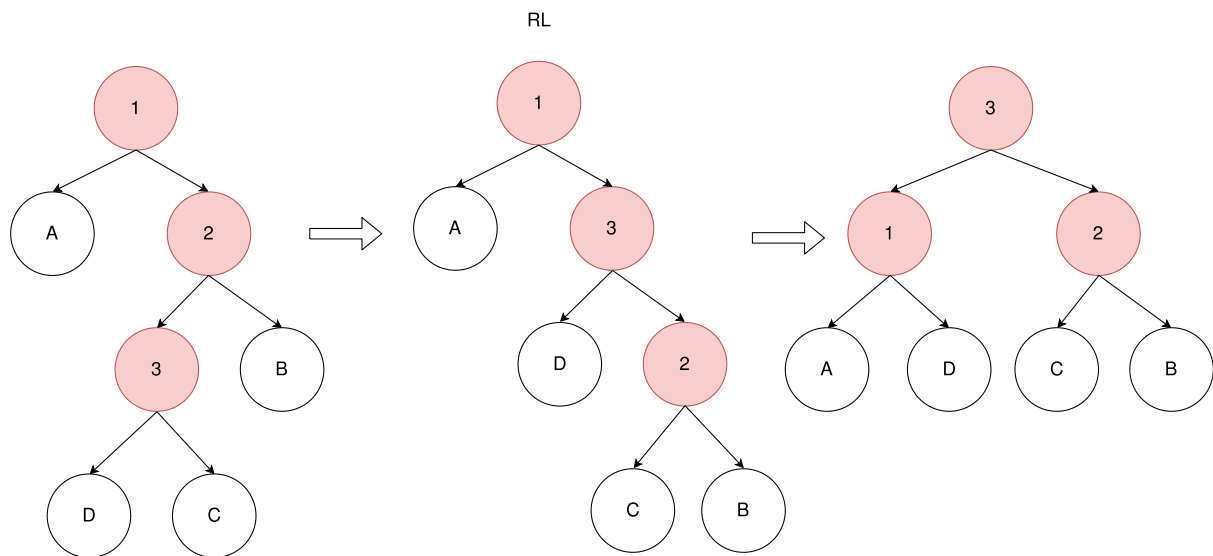


RR



LR





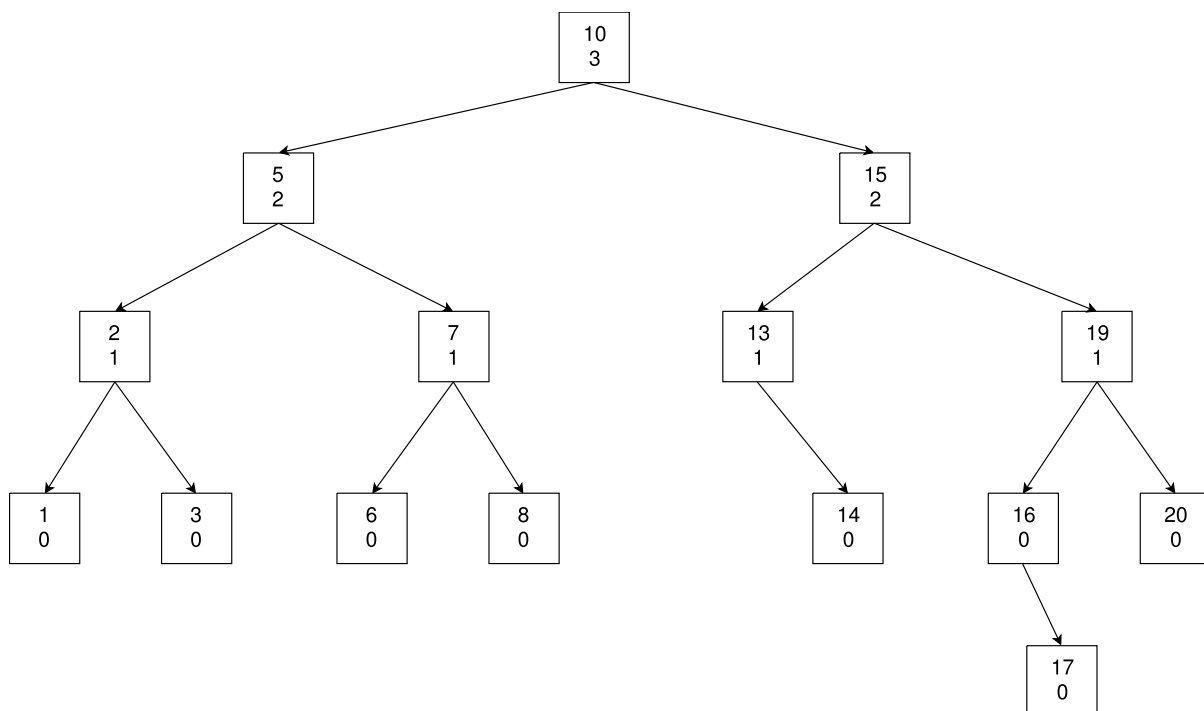
上面四种情况包含了所有从不平衡转化为平衡的步骤，其中单向右旋平衡处理LL，单向左旋平衡处理RR，双向旋转（先左后右）平衡处理LR，双向旋转（先右后左）平衡处理RL。

这四种操作既能够平衡左右子树的高度，还能够保持树的有序性。即平衡后的树的左子树中所有节点仍然小于（或大于）树的根节点，而右子树中所有节点仍然大于（或小于）树的根节点。

AVL树的每个节点都有一个高度值 $depth$ ，树的平衡因子为 $balanceFactor = leftTree.depth - rightTree.depth$ ，即左右子树的深度之差。当一个节点的 $|balanceFactor| \leq 1$ 时该子树平衡；当 $|balanceFactor| \leq 2$ 时该子树不平衡。

将空节点的高度值视作 -1 ，一个节点的高度值为 $depth\{node\} = \max(depth\{node.leftChild\}, depth\{node.rightChild\}) + 1$ 。上面LL、RR、LR和RL四种操作，都会将其节点 1 的高度值减 2，其余节点的高度值都不变。

对于下面这个AVL树，每个节点中上面的数字是节点下标号，下面的数字是该节点的高度值 $depth$ 。将节点 18 插入下面的AVL树：



(1) 从根节点开始，将节点 18 与节点 10 比较，有 $18 \leq 10$ ，因此把节点 18 插入节点 10 的右子树；

(2) 将节点 18 与节点 15 比较，有 $18 \leq 15$ ，因此把节点 18 插入节点 15 的右子树；

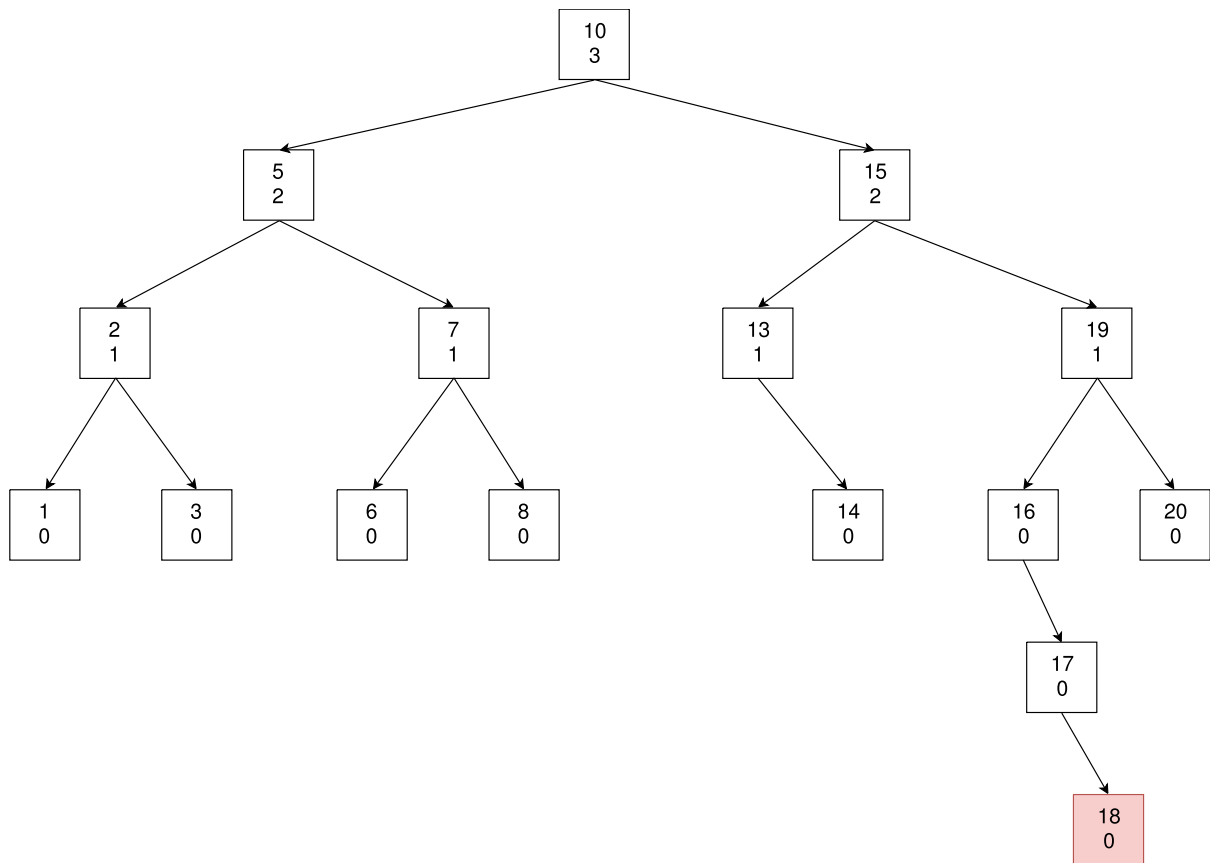
(3) 将节点 18 与节点 19 比较，有 $18 \leq 19$ ，因此把节点 18 插入节点 19 的左子树；

(4) 将节点 18 与节点 16 比较，有 $18 \leq 16$ ，因此把节点 18 插入节点 16 的右子树；

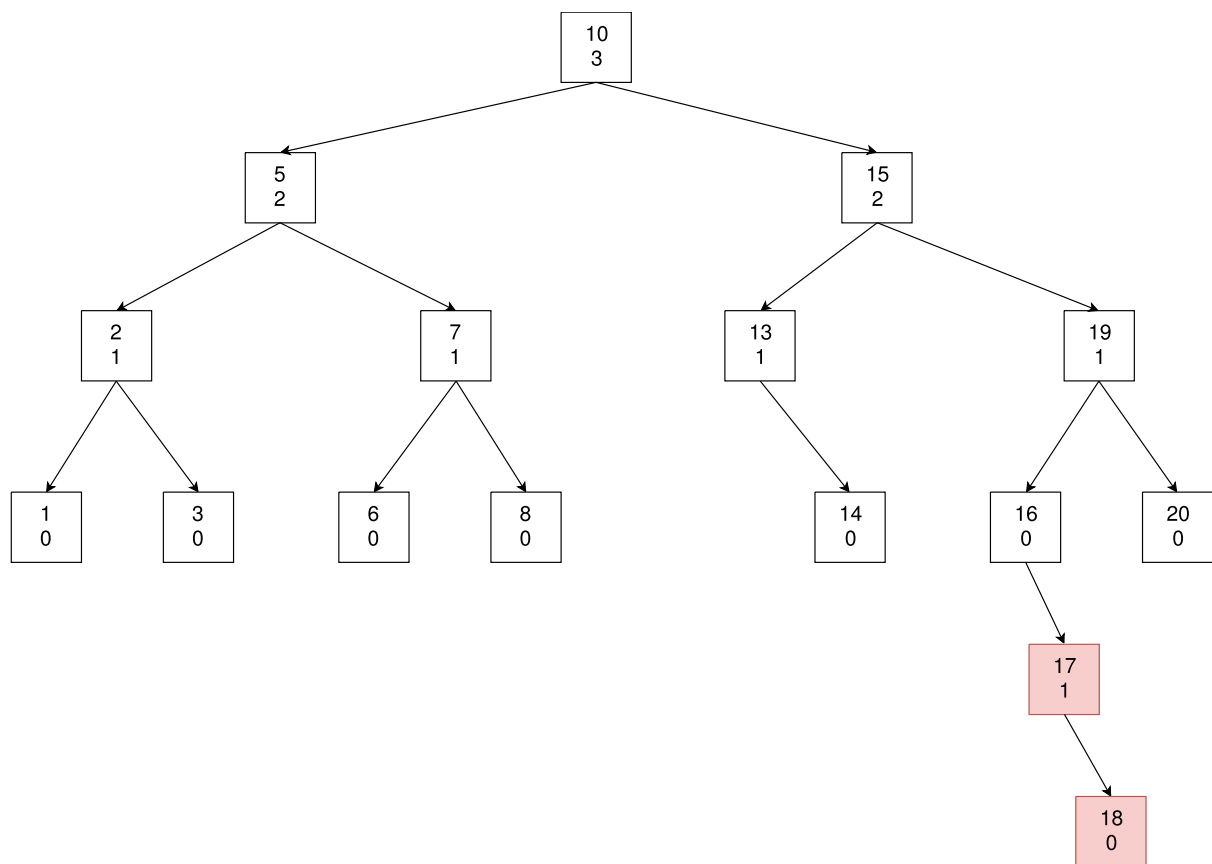
(5) 将节点 18 与节点 17 比较，有 $18 \leq 17$ ，因此把节点 18 插入节点 17 的右子树，节点 17 的右孩子节点为空，因此节点 18 成为节点 17 的右孩子节点；

然后从节点 18 开始，向上依次更新所有节点的高度值，若新的高度值不满足AVL树的平衡性，则进行旋转操作：

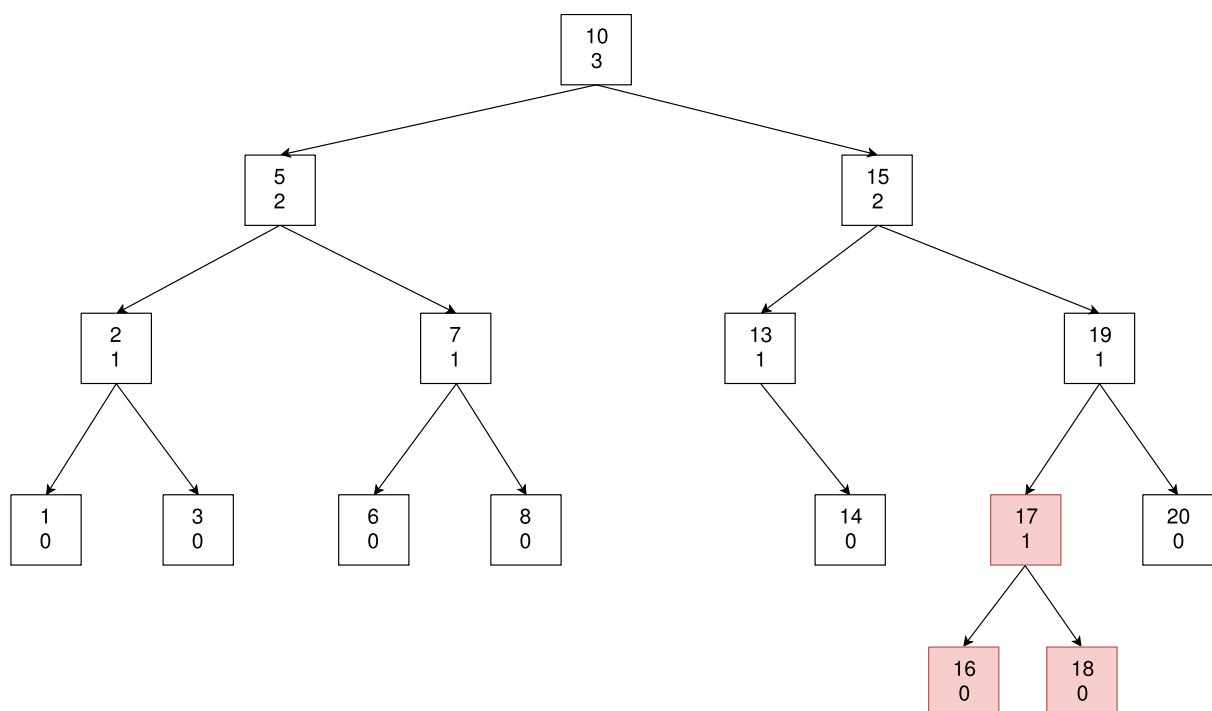
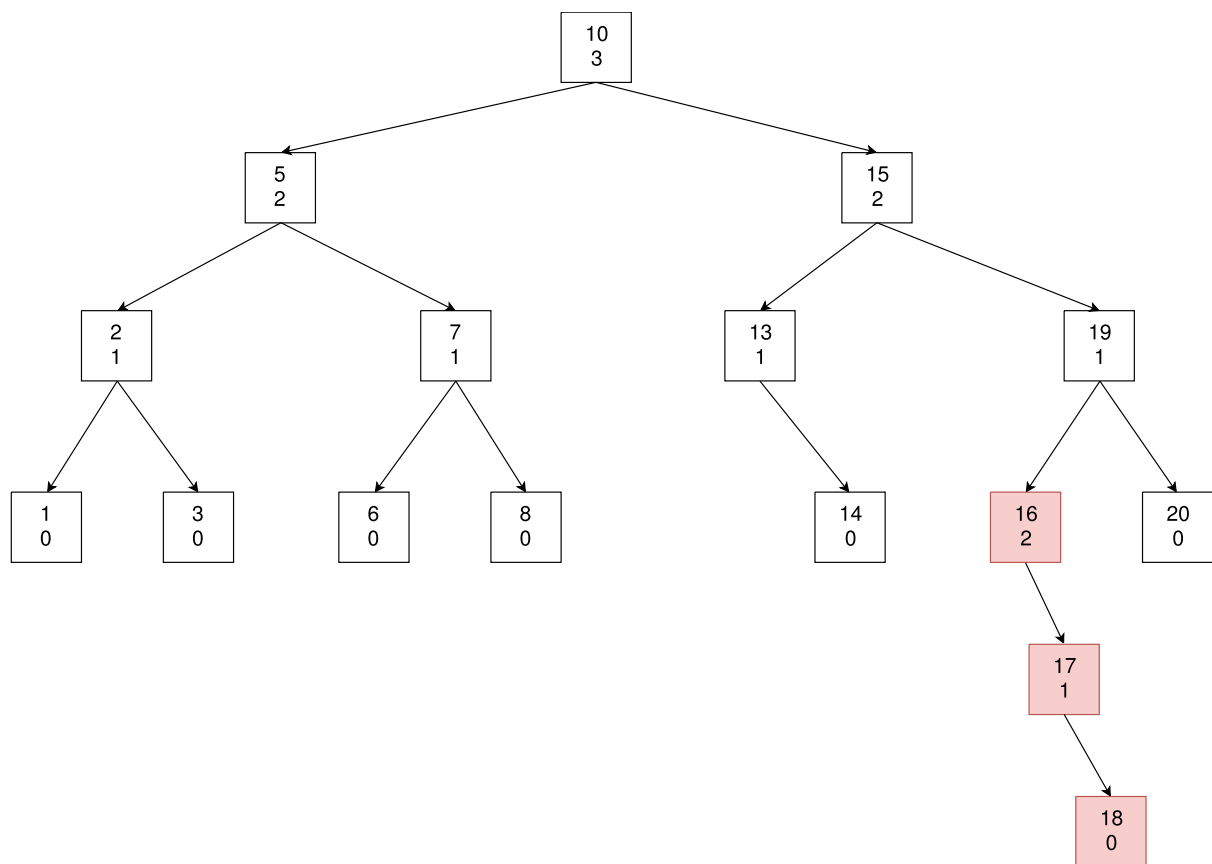
(6) 节点 18 为叶子节点，因此高度值为 0；



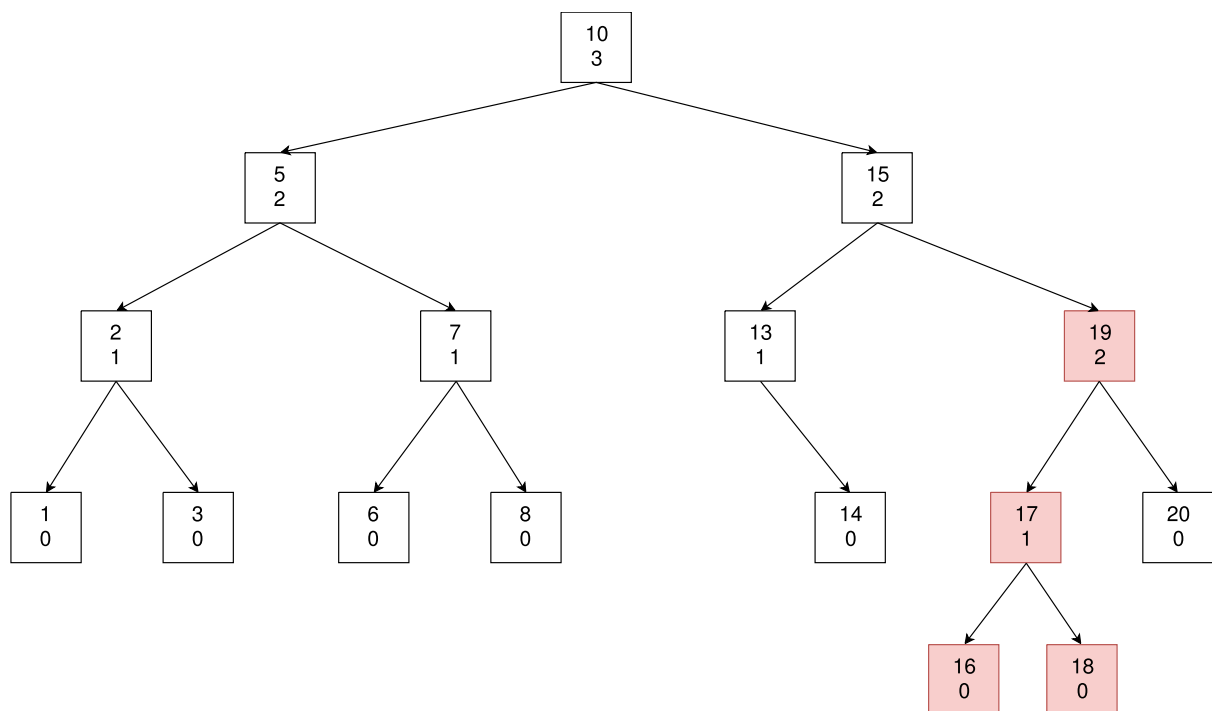
(7) 节点 17 的平衡因子为 $\text{balanceFactor}\{17\} = |\text{depth}\{\text{nil}\} - \text{depth}\{18\}| = |-1 - 0| = 1$, 不需要旋转, 高度值更新为 $\text{depth}\{17\} = \max(\text{depth}\{17.\text{leftChild}\}, \text{depth}\{17.\text{rightChild}\}) + 1 = \max(-1, 0) + 1 = 1$;



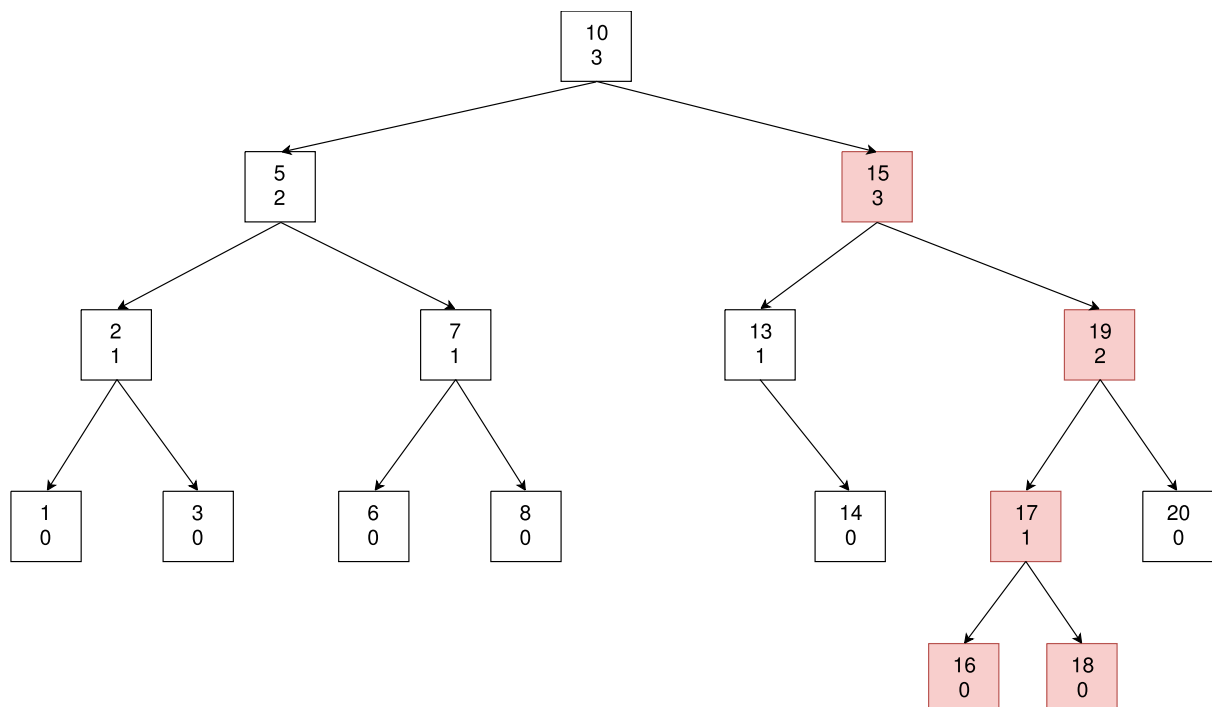
(8) 节点 16 的平衡因子为 $\text{balanceFactor}\{16\} = |\text{depth}\{\text{nil}\} - \text{depth}\{17\}| = |-1 - 1| = 2$ ，高度值更新为 $\text{depth}\{16\} = \max(\text{depth}\{16.\text{leftChild}\}, \text{depth}\{16.\text{rightChild}\}) + 1 = \max(-1, 1) + 1 = 2$ ，由于节点 16 的平衡因子超过 1，需要进行RR操作，旋转后节点 16 的高度值减 2；



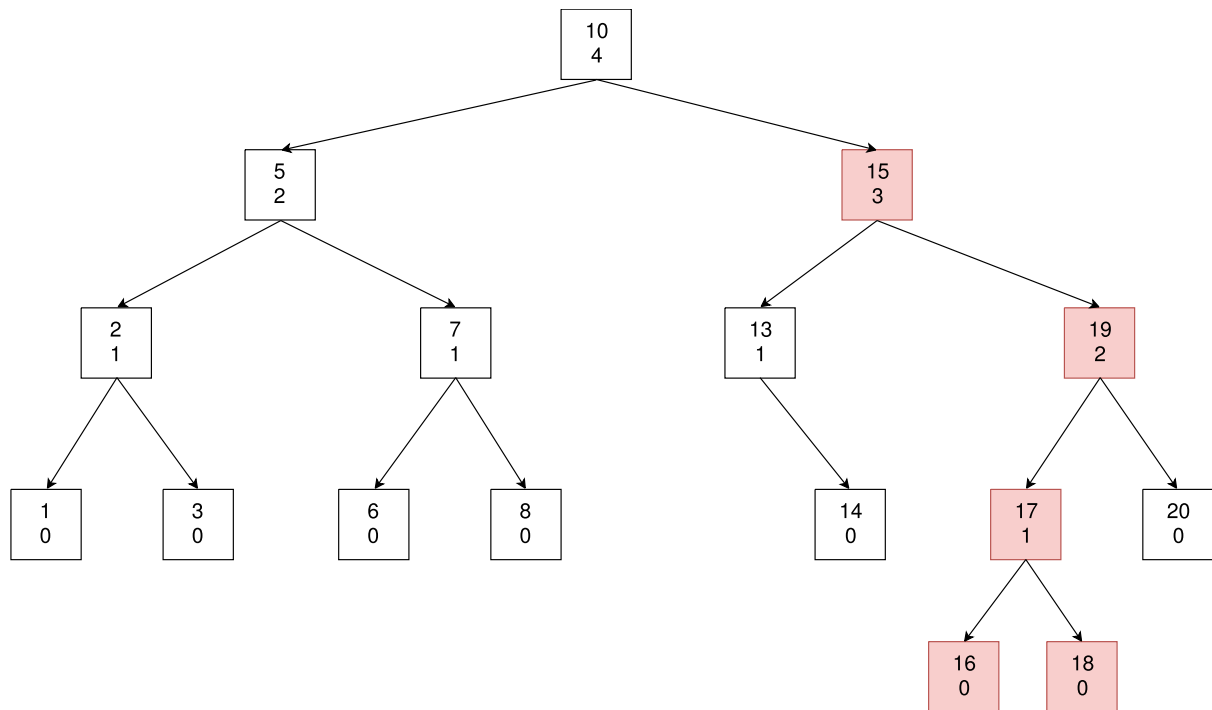
(9) 节点 19 的平衡因子为 $\text{balanceFactor}\{19\} = |\text{depth}\{17\} - \text{depth}\{20\}| = |1 - 0| = 1$, 高度值更新为 $\text{depth}\{19\} = \max(\text{depth}\{19.\text{leftChild}\}, \text{depth}\{19.\text{rightChild}\}) + 1 = \max(1, 0) + 1 = 2$;



(10) 节点 15 的平衡因子为 $\text{balanceFactor}\{15\} = |\text{depth}\{13\} - \text{depth}\{19\}| = |1 - 2| = 1$, 高度值更新为 $\text{depth}\{15\} = \max(\text{depth}\{15.\text{leftChild}\}, \text{depth}\{15.\text{rightChild}\}) + 1 = \max(1, 2) + 1 = 3$;



(11) 节点 10 的平衡因子为 $\text{balanceFactor}\{10\} = |\text{depth}\{5\} - \text{depth}\{15\}| = |2 - 3| = 1$, 高度值更新为 $\text{depth}\{10\} = \max(\text{depth}\{10.\text{leftChild}\}, \text{depth}\{10.\text{rightChild}\}) + 1 = \max(2, 3) + 1 = 4$;



源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

RedBlackTree 红黑树

- [Red Black Tree - 红黑树](#)
 - [描述](#)
 - [源码](#)
 - [测试](#)

Red Black Tree - 红黑树

描述

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

Chapter-4 DynamicProgramming 第4章 动态规划

- 第4章 动态规划

第4章 动态规划

1. [KnowledgePoint](#) 知识要点
2. [LinearDP](#) 线性动态规划
 - i. [LongestCommonSubsequence](#) 最长公共子序列
 - ii. [LongestIncreasingSubsequence](#) 最长递增子序列
 - iii. [LongestIncreasingSubsequenceExtension](#) 最长递增子序列扩展
 - iv. [BidirectionalSubsequence](#) 双向子序列
3. [KnapsackDP](#) 背包问题
 - i. [ZeroOneKnapsack](#) 01背包
 - ii. [ZeroOneKnapsackExtension](#) 01背包扩展
 - iii. [CompleteKnapsack](#) 完全背包
 - iv. [TwoDimensionKnapsack](#) 二维背包
 - v. [GroupKnapsack](#) 分组背包
4. [RegionalDP](#) 区域动态规划
 - i. [MinimumMergeCost](#) 最小合并代价
 - ii. [MinimumMergeCostExtension](#) 最小合并代价扩展
 - iii. [MaximumBinaryTreeMerge](#) 最大二叉树合并
5. [TreeDP](#) 树形动态规划
 - i. [BinaryTreeDP](#) 二叉树动规
 - ii. [MultipleTreeDP](#) 多叉树动规
 - iii. [MultipleTreeDPExtension](#) 多叉树动规问题扩展
 - iv. [LoopedMultipleTreeDP](#) 带环多叉树动规
 - v. [TraverseBinaryTreeDP](#) 遍历二叉树动规

KnowledgePoint 知识要点

- Knowledge Point - 知识要点
 - 运筹学

Knowledge Point - 知识要点

动态规划 (Dynamic Programming, DP) 是运筹学 (线性规划、网络流等问题也属于运筹学) 中的一个问题分支, 用于求解最优解。

DP模型基本上是一种递归方程, 把问题的各阶段联系起来, 保证每个阶段的最优可行解对于整个问题既是最优的也是可行的。一般来说递归方程的结构对于初学者并不“合乎逻辑”, “难以理解”, 最好的做法是做一些适当的计算来理解方程的正确性。运筹学中将动态规划问题分为确定性动态规划和随机性动态规划。在我们的大学生计算机算法中, 只考虑确定性动态规划问题。

DP计算的基本特性:

- (1) 每个阶段所做的计算都是该阶段可行路径的函数, 并且只针对该阶段;
- (2) 当前阶段仅仅连接到紧接着的上一阶段, 与再前面的阶段无关。这种连接是以最短距离小结的形式表示出上一阶段的输出;

递归公式: $x_i = f(x_{i-1})$, 其中 x_0 为初始的常数, i 从 1 开始增加。在这个方程中, x_i 是系统在阶段 i 的状态, 它只由前一个状态 x_{i-1} 计算得到, 而不需要考虑更前一个状态 x_{i-2} 和其他之前的状态, 而后一个状态 x_{i+1} 也只需要 x_i 的信息。状态拆分使得我们只需要根据当前状态就可以对后一个状态作出最优决策, 而不需要重新考察所有前面阶段所做的决策。递归公式也称作状态转移方程。

最优性原则: 对以后阶段所做出的未来决策会产生一个最优策略, 它与前面各阶段所采用的策略无关。

DP模型具有 3 个基本要素:

- (1) 定义阶段;
- (2) 定义每个阶段的可选方案;
- (3) 定义每个阶段的状态;

在这 3 个要素中, 状态的定义往往是最微妙的: 是什么样的关系把各阶段联系在一起; 在当前阶段作出可行决策, 又不用重新考察前面阶段所做的决策, 需要什么信息。

实际编程中, 状态通常会存储在数组中, 对于长度为 n 的数组, 其范围不再是 $[0, n-1]$, 而会

专门把 0 空出来，范围是 $[1, n]$ ，0 这个部分一般用来存储固定的初始值。

本书中我们将DP问题分为 4 个部分：

- (1) 线性DP；
- (2) 背包问题；
- (3) 区域DP；
- (4) 树型DP；

运筹学

- https://en.wikipedia.org/wiki/Operations_research
- <https://book.douban.com/subject/4747771/>

Section-2 KnapsackDP 第2节 背包问题

ZeroOneKnapsack 01背包

- [Zero One Knapsack - 01背包](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Zero One Knapsack - 01背包

问题

你面前摆放着 n 个珠宝（共 n 种，每种 1 个），已知珠宝 s_i 的价值是 v_i ，重量是 w_i 。给你一个背包，你可以自由挑选珠宝装到背包中，但背包可以装载的最大重量为 t 。求背包能够装载珠宝的最大价值 v 。

解法

设 $f(i, j)$ 为背包中放入前 i 件物品，重量不大于 j 的最大价值，其中 $i \in [1, n]$ ， $j \in [0, t]$ 。有如下状态转移方程：

$$f(i, j) = \begin{cases} 0 & \text{(初始化)} \\ f(i-1, j) & \text{if } i, j > 0, j < w_i \\ \max(f(i-1, j), f(i-1, j-w_i)+v_i) & \text{if } i, j > 0, j \geq w_i \end{cases}$$

(1) 用数组中的下标 0 来存储初始的固定值，背包中没有放入任何珠宝时， $f(0, j) = 0$ ；

(2) 对于第 i 件珠宝 s_i ，若背包中还能够装载的重量大于 w_i ，那么装入背包，则价值增大 v_i ，剩余重量（还能装载的重量）减小 w_i ，即 $f(i, j) = f(i-1, j-w_i)+v_i$ ；若不装入背包，则一切维持不变，即 $f(i, j) = f(i-1, j)$ 。选择这两种情形中的最大值；

$f(n, t)$ 即为 n 个珠宝中重量不超过 t 的最大价值。该算法的时间复杂度是 $O(n \times t)$ 。

源码


```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

ZeroOneKnapsackExtension 01背包扩展

- [Zero One Knapsack Extension - 01背包扩展](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Zero One Knapsack Extension - 01背包扩展

问题

在<Zero One Knapsack>的基础上，不仅求出最大价值，还求出具体选择了哪些珠宝，即求出具体选择方案。

解法

仍然按照<Zero One Knapsack>中的方法，设 $f(i, j)$ 为背包中放入前 i 件物品，重量不大于 j 的最大价值，其中 $i \in [1, n]$ ， $j \in [0, t]$ 。有如下状态转移方程：

$$f(i, j) = \begin{cases} 0 & \text{(初始化)} \\ f(i-1, j) & \text{if } i, j \leq 0 \\ \max(f(i-1, j), f(i-1, j-w_i)+v_i) & \text{if } i, j > 0, j \geq w_i \end{cases}$$

额外的，设 $g(i, j, k)$ 表示重量不大于 j ，最大价值为 k ，第 i 件珠宝是否被装入背包，其中 $i \in [1, n]$ ， $j \in [0, t]$ ， $k \in [0, \sum\{v_i\}]$ 。若 $g(i, j, k) = \text{true}$ 则该珠宝被选中；若 $g(i, j, k) = \text{false}$ 则该珠宝未被选中。在遍历所有珠宝的过程中，可以求出所有的 $f(i, j)$ 和 $g(i, j, k)$ 。

已知当前背包的总价值为 v ，总重量为 t 。逆向的从最后一个珠宝 n 开始，对于第 i 件珠宝，若 $g(i, t, v) = \text{true}$ 则说明珠宝 i 被装入了背包，那么 $v = v - v_i$ ， $t = t - w_i$ ，然后继续考虑下一件珠宝 $i-1$ 。

该算法的时间复杂度是 $O(n \times t)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

CompleteKnapsack 完全背包

- [Complete Knapsack - 完全背包](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Complete Knapsack - 完全背包

问题

你面前摆放着 n 种珠宝，每种都有无穷多 $+\infty$ 个，已知珠宝 s_i 的价值是 v_i ，重量是 w_i 。给你一个背包，你可以自由挑选珠宝装到背包中，但背包可以装载的最大重量为 t 。求背包能够装载珠宝的最大价值 v 。

解法

设 $f(i, j)$ 为背包中放入前 i 件物品，重量不大于 j 的最大价值，其中 $i \in [1, n]$ ， $j \in [0, t]$ 。有如下状态转移方程：

$$f(i, j) = \begin{cases} 0 & \text{(初始化)} i \in [0, n], j \in [0, t] \\ \max(f(i-1, j), f(i-1, j-k \times w_i) + k \times v_i) & i, j > 0, k \geq 0, j \geq k \times w_i \end{cases}$$

(1) 将 $f(i, j)$ 全部初始化为 0；

(2) 对于第 i 件珠宝 s_i ，背包的剩余重量（还能装载的重量）为 w ，可以装进 k 个该珠宝（其中 $k \geq 0$ ，且 $w \geq k \times w_i$ ），那么背包的价值增大 $k \times v_i$ ，剩余重量减小 $k \times w_i$ ，即 $f(i, j) = f(i-1, j-k \times w_i) + k \times v_i$ ；若不装入背包，则一切维持不变，即 $f(i, j) = f(i-1, j)$ 。选择这两种情形中的最大值；

$f(n, t)$ 即为 n 个珠宝中重量不超过 t 的最大价值。该算法的时间复杂度是 $O(n \times t^2)$ ，因为状态转移方程中的参数 k 的规模与背包最大重量 t 线性相关。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



TwoDimensionKnapsack 二维背包

- [Two Dimension Knapsack - 二维背包](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Two Dimension Knapsack - 二维背包

问题

你面前摆放着 n 个珠宝（共 n 种，每种 1 个），已知珠宝 s_i 的价值是 v_i ，重量 1 是 $w1_i$ ，重量 2 是 $w2_i$ 。给你一个背包，你可以自由挑选珠宝装到背包中，但背包可以装载的最大重量 1 为 $t1$ ，最大重量 2 为 $t2$ 。求背包能够装载珠宝的最大价值 v 。

该问题与01背包的区别就是，重量属性变成了 2 维属性，背包中所有珠宝的总重量 1 不能超过 $t1$ ，总重量 2 不能超过 $t2$ 。

解法

设 $f(i, j, k)$ 为背包中放入前 i 件物品，重量 1 不大于 j ，重量 2 不大于 k 的最大价值，其中 $i \in [1, n]$ ， $j \in [0, t1]$ ， $k \in [0, t2]$ 。有如下状态转移方程：

$$\begin{aligned}
 f(i, j, k) = & \\
 & \begin{cases}
 0 & \text{(初始化)} \\
 i = 0 \setminus \\
 f(i-1, j, k) & \text{if } i, j, k \leq 0 \setminus \\
 \max\{f(i-1, j, k), f(i-1, j-w1_i, k-w2_i)+v_i\} & \text{if } i, j, k > 0, j \geq w1_i, k \geq w2_i
 \end{cases}
 \end{aligned}$$

用数组中的下标 0 来存储初始的固定值，背包中没有放入任何珠宝时， $f(0, j) = 0$ ；

对于第 i 件珠宝 s_i ，背包的剩余重量 1（还能装载的重量）为 $w1$ ，剩余重量 2 为 $w2$ ，若 $w1 \geq k \times w1_i$ ， $w2 \geq k \times w2_i$ ，那么可以装进 1 个珠宝 s_i ，背包的价值增大 v_i ，剩余重量 1 减小 $w1_i$ ，剩余重量 2 减小 $w2_i$ 。即 $f(i, j, k) = f(i-1, j-w1_i, k-w2_i)+v_i$ ；若不装入背包，则一切维持不变，

即 $f(i, j, k) = f(i-1, j, k)$ 。选择这两种情形中的最大值；

$f(n, t_1, t_2)$ 即为 n 个珠宝中重量 1 不超 t_1 ，重量 2 不超过 t_2 的最大价值。该算法的时间复杂度是 $O(n \times t_1 \times t_2)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

GroupKnapsack 分组背包

- [Group Knapsack - 分组背包](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Group Knapsack - 分组背包

问题

你面前摆放着 n 个珠宝（共 n 种，每种 1 个），这些珠宝被分成 m 个组（显然 $n \geq m$ ）。已知珠宝 s_i 的价值是 v_i ，重量是 w_i 。给你一个背包，你可以挑选珠宝装到背包中，但背包可以装载的最大重量为 t ，并且同一个组的珠宝至多只能选择 1 个。求背包能够装载珠宝的最大价值 v 。

该问题与01背包的区别就是，对珠宝进行了分组，并且一个组内的珠宝互斥。

解法

设 $f(i, j)$ 为背包中放入前 i 组物品，重量不大于 j 的最大价值，其中 $i \in [1, m]$ ， $j \in [0, t]$ 。第 i 组中有 $group_i$ 个珠宝，其中某珠宝 k 的价值是 v_k ，重量是 w_k 。则有如下状态转移方程：

$$f(i, j) = \begin{cases} 0 & \text{(\text{初始化}) } i = 0 \\ f(i-1, j) & \text{ } i, j > 0 \\ \max(f(i-1, j), f(i-1, j - w_k) + v_k) & \text{ } i, j > 0, k \in [1, group_i], j \geq w_k \end{cases}$$

(1) 用数组中的下标 0 来存储初始的固定值，背包中没有放入任何珠宝时， $f(0, j) = 0$ ；

(2) 对于第 i 组珠宝，背包的剩余重量（还能装载的重量）为 w ，在第 i 组珠宝中选择某个珠宝 k ，若 $w \geq w_k$ ，那么可以装进珠宝 k ，背包的价值增大 v_k ，剩余重量减小 w_k ，即 $f(i, j) = f(i-1, j - w_k) + v_k$ ；若不装入背包，则一切维持不变，即 $f(i, j) = f(i-1, j)$ 。选择这两种情形中的最大值；

$f(m, t)$ 即为 m 组珠宝中重量不超过 t 的最大价值。该算法的时间复杂度是 $O(n \times t)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

Section-3 RegionalDP 第3节 区域动规

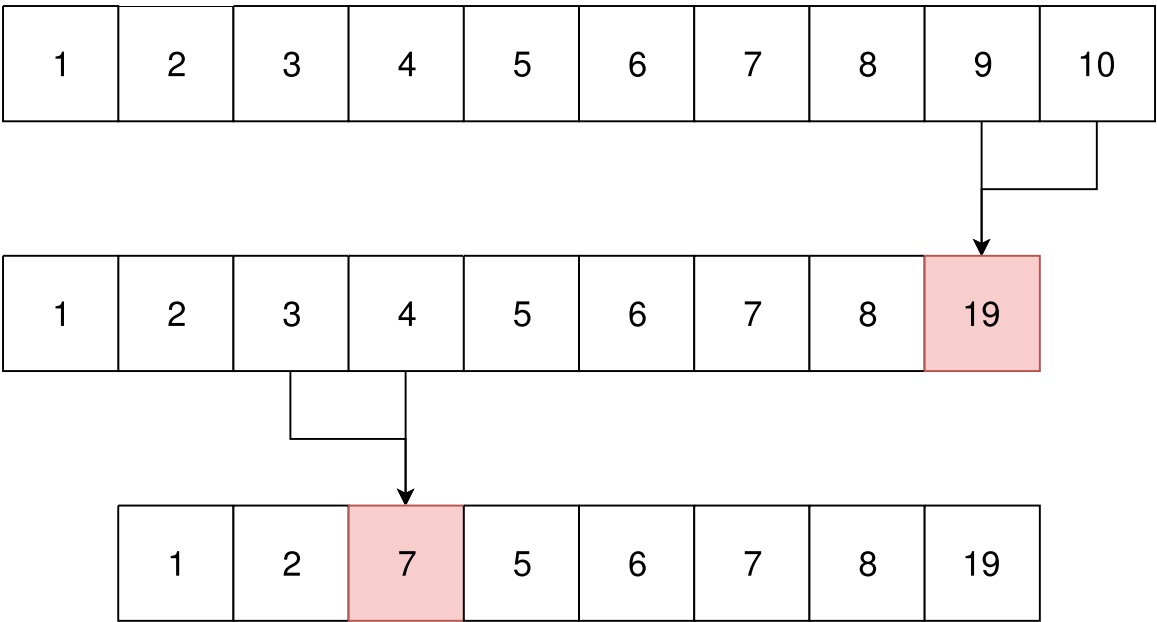
MinimumMergeCost - 最小合并代价

- [Minimum Merge Cost - 最小合并代价](#)
 - [问题](#)
 - [解法](#)
 - [石子合并](#)
 - [源码](#)
 - [测试](#)

Minimum Merge Cost - 最小合并代价

问题

对长度为 n 的序列 s 进行合并，每次将相邻的两个元素 a 和 b 合并为一个新的元素 c ，并且 $c = a+b$ ，合并产生的代价也为 $a+b$ 。经过 $n-1$ 次合并后，序列 s 被合并为 1 个数字，这个过程的代价是之前所有合并的代价总和。求出将序列 s 合并为一个数字的最小合并代价。合并过程如图：



.....

本问题的原型为“石子合并”。

解法

设 $\text{sum}(i, j)$ 为序列中区域 $s[i, j]$ 的所有元素之和, 设 $f(i, j)$ 为合并区域 $s[i, j]$ 产生的最小代价, 其中 $i, j \in [1, n]$ 且 $i \leq j$ 。因此有如下状态转移方程:

$$f(i, j) = \begin{cases} 0 & \text{(初始化)} i, j \in [0, n], i = j \\ +\infty & \text{(初始化)} i, j \in [0, n], i \neq j \\ \min \{f(i, k) + f(k+1, j) + \text{sum}(i, k) + \text{sum}(k+1, j)\} & i, j, k \in [1, n], i \leq k \leq j \end{cases}$$

(1) $s[i, i]$ 不需要合并, 因此 $f(i, i) = 0$;

(2) $s[i, j]$ 需要合并, 我们的最终目标是获取合并最小代价, 因此设未知的 $f(i, j) = +\infty$;

(3) 假设将 $s[i, k]$ 和 $s[k+1, j]$ 这两个区域的元素合并。合并 $s[i, k]$ 和 $s[k+1, j]$ 的过程中, 已知 $s[i, k]$ 范围的总和为 $\text{sum}(i, k)$, 消耗的代价为 $f(i, k)$, $s[k+1, j]$ 范围的总和为 $\text{sum}(k+1, j)$, 消耗的代价为 $f(k+1, j)$ 。因为 $k \in [i, j]$, 因此 $f(i, j) = \min \{ f(i, k) + f(k+1, j) + \text{sum}(i, k) + \text{sum}(k+1, j) \}$, 选择该范围中所有结果的最小值即可;

$f(0, n)$ 即为序列 s 的最小合并代价。该算法的时间复杂度是 $O(n^2)$ 。

石子合并

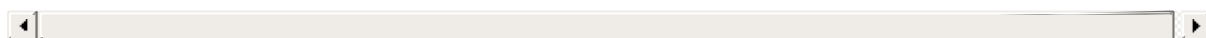
- <http://acm.nyist.edu.cn/JudgeOnline/problem.php?pid=737>

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



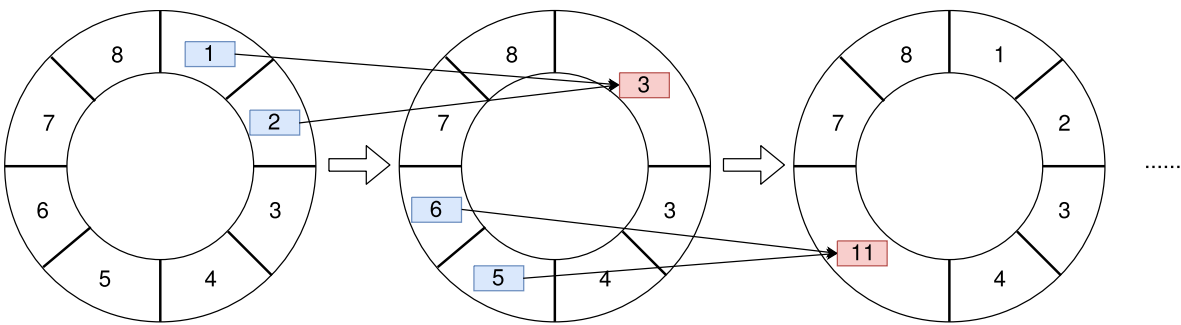
MinimumMergeCostExtension - 最小合并代价扩展

- [Minimum Merge Cost Extension - 最小合并代价扩展](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Minimum Merge Cost Extension - 最小合并代价扩展

问题

在<Minimum Merge Cost>问题的基础上进行变化，序列 s 是头尾相接的，仍然求最小合并代价。下图演示了一个合并过程：



解法

本问题与<Minimum Merge Cost>问题的核心区别在于序列是首尾相接的，取巧的办法就是把长度为 n 的序列 s 扩展为原始的 2 倍长度，多出的部分用 s 再填充一遍，则有 $s[j] = s[i]$ ，其中 $i \in [1, n]$ ， $j \in [n+1, 2n]$ 且 $j = i+n$ ，在 $s[n]$ 和 $s[n+1]$ 两个相邻元素的位置可以模拟出首尾相接的效果。而状态转移方程完全不变，只需要把算法的范围调整为 $[0, 2n]$ 即可。

设 $\text{sum}(i, j)$ 为序列中区域 $s[i, j]$ 的所有元素之和，设 $f(i, j)$ 为合并区域 $s[i, j]$ 产生的最小代价，其中 $i, j \in [1, 2n]$ 且 $i \leq j$ 。因此有如下状态转移方程：

$$f(i, j) = \begin{cases} 0 & (\text{初始化}) \\ i, j \in [0, 2n], i = j \end{cases}$$

$$+\infty \text{ \& (初始化) } i, j \in [0, 2n], i \neq j \text{ \& } \min \{ f(i, k) + f(k+1, j) + \text{sum}(i, k) + \text{sum}(k+1, j) \} \text{ \& } i, j \in [0, 2n], i \leq k \leq j \text{ \& } \end{cases}$$

(1) $s[i, i]$ 不需要合并，因此 $f(i, i) = 0$ ；

(2) $s[i, j]$ 需要合并，我们的最终目标是获取合并最小代价，因此设未知的 $f(i, j) = +\infty$ ；

(3) 假设将 $s[i, k]$ 和 $s[k+1, j]$ 这两个区域的元素合并。合并 $s[i, k]$ 和 $s[k+1, j]$ 的过程中，已知 $s[i, k]$ 范围的总和为 $\text{sum}(i, k)$ ，消耗的代价为 $f(i, k)$ ， $s[k+1, j]$ 范围的总和为 $\text{sum}(k+1, j)$ ，消耗的代价为 $f(k+1, j)$ 。因为 $k \in [i, j]$ ，因此 $f(i, j) = \min \{ f(i, k) + f(k+1, j) + \text{sum}(i, k) + \text{sum}(k+1, j) \}$ ，选择该范围中所有结果的最小值即可；

$f(0, 2n)$ 即为序列 s 的最小合并代价。该算法的时间复杂度是 $O(n^2)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



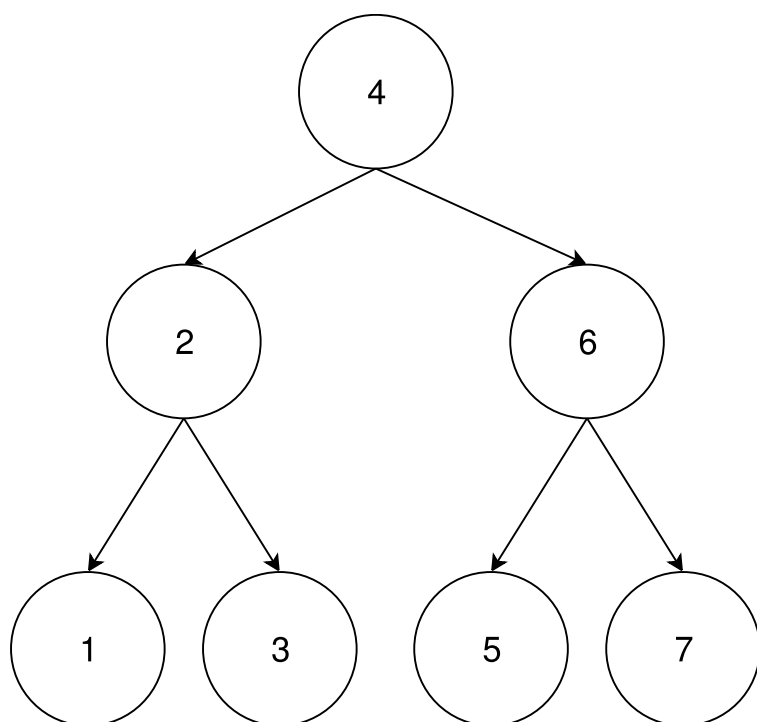
MaximumBinaryTreeMerge - 最大二叉树合并

- [Maximum Binary Tree Merge - 最大二叉树合并](#)
 - [问题](#)
 - [解法](#)
 - [加分二叉树](#)
 - [源码](#)
 - [测试](#)

Maximum Binary Tree Merge - 最大二叉树合并

问题

拥有 n 个节点的二叉树，按照中序遍历将所有节点标记为 $[1, n]$ ，如图：



节点 i 拥有价值 v_i ，将子树进行合并，产生的代价的计算方法是 $v_{\{tree\}} = v_{\{leftChild\}} \times v_{\{rightChild\}} + v_{\{root\}}$ ，即其左子树的合并代价乘以右子树的合并代价，再加根节点自身的价值，特别的我们规定空子树的合并代价为 1。合并顺序的不同会使最终整个树的合并代价不同，求该二叉树的最大合并代价。

本问题的原型为“加分二叉树”。

解法

将二叉树中的所有节点按照中序遍历依次编号为 $[1, n]$ ，根据中序遍历的性质，可知连续节点 $[i, j]$ 刚好属于 1 个子树，且在 $[i, j]$ 中选取节点 k 作为根节点 ($i \leq k \leq j$)，则其左子树为 $[i, k-1]$ ，右子树范围为 $[k+1, j]$ 。例如上图中， $[1, 3]$ 属于子树 2 (以 2 为根节点的子树)， $[5, 7]$ 属于子树 6。设 $f(i, j)$ 为以节点 $[i, j]$ 组成的子树的最大合并代价，其中 $i, j \in [1, n]$ 且 $j \leq i$ ，其转移方程如下：

$$f(i, j) = \begin{cases} 1 & \text{(初始化)} i, j \in [0, n] \setminus [1, n], i = j \\ \max \{ f(i, k-1) + f(k+1, j) + v_k \mid i \leq k \leq j \} & \text{其他情况} \end{cases}$$

(1) 将所有可能情况都初始化为最小的合并代价，即 1；

(2) 对于只有一个节点的子树来说，其合并代价为自身根节点的价值加 1，即 $f(i, i) = 1 + v_i$ ，因为左右子树都是空子树，其合并代价为 1；

(3) 将 $f(i, j)$ 分为 $f(i, k-1)$ 和 $f(k+1, j)$ 左右两个子树，则 $f(i, j) = f(i, k-1) + f(k+1, j) + v_k$ ，其中 $i \leq k \leq j$ 。在 $[i, j]$ 范围内遍历所有情况，选取最大的即可；

$f(1, n)$ 即为二叉树的最大合并价值。该算法的时间复杂度是 $O(n^2)$ 。

加分二叉树

- <http://codevs.cn/problem/1090/>

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



Section-4 TreeDP 第4节 树形动规

BinaryTreeDP 二叉树动规

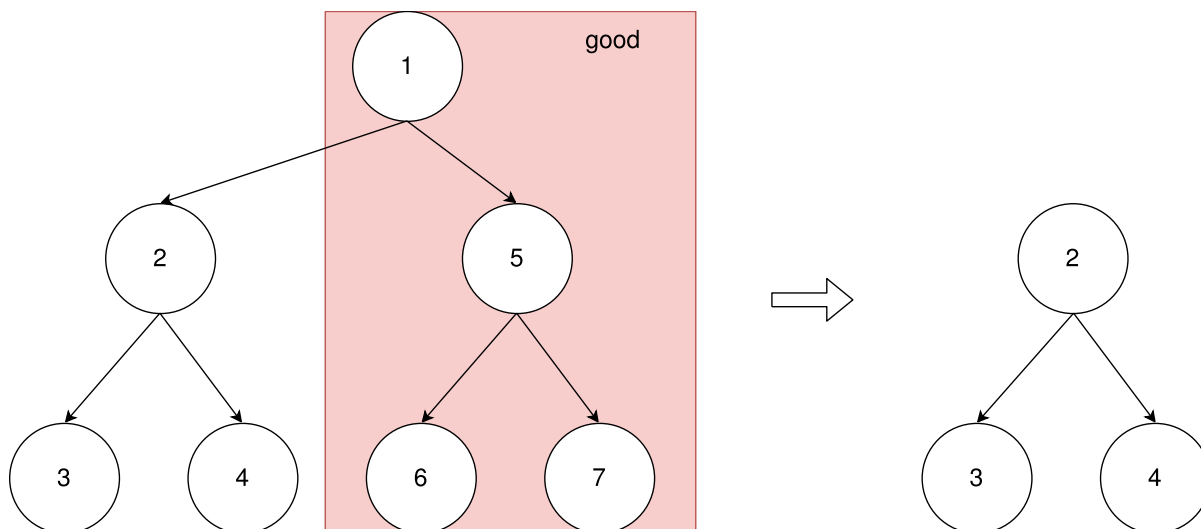
- [Binary Tree DP - 二叉树动规](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Binary Tree DP - 二叉树动规

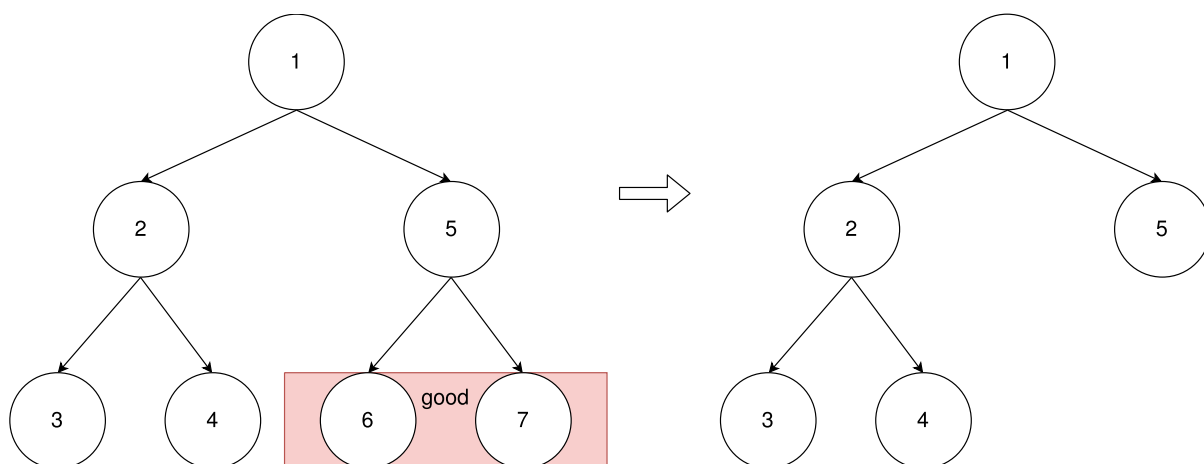
问题

拥有 n 个节点的二叉树，节点下标范围为 $[0, n)$ ，节点 i 的权值为 v_i ($v_i > 0$)，整个二叉树的权值为所有节点的权值之和。现在要求只保留 m 个节点 ($0 < m < n-1$)，剪裁掉的节点数量为 $n-1-m$ ，要求剩余部分仍然是一个二叉树，而不能是多个二叉树。如图：

(1) 正确剪裁



(2) 正确剪裁



(3) 错误剪裁

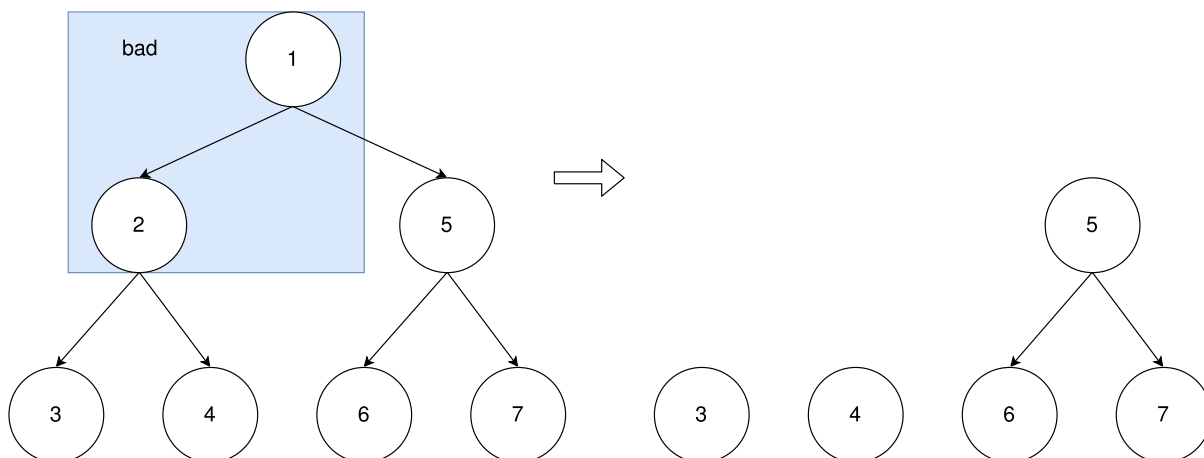


图 (1) 和 (2) 剪裁后的剩余部分仍然是二叉树，图 (3) 剪裁后的剩余部分分为了 3 个部分。对于拥有 n 个节点的二叉树，求出保留 m 个节点的二叉树的最大权值。

解法

设 $f(i, j)$ 表示以节点 i 为根节点的树上，保留 j 个节点（包括节点 i 自己）的最大权值。其转移方程如下：

$$f(i, j) = \begin{cases} v_i & \text{(初始化) } i, j \in [0, n], i = j \\ \max\{f(\text{leftChild}_i, k) + f(\text{rightChild}_i, j-1-k) + v_i\} & i, j \in [0, n], i \neq j \end{cases}$$

(1) 节点数量为 1 的二叉树，其最大权值即为节点自己的权值，即 $f(i, i) = v_i$ ；

(2) 对于该二叉树的左右子树，其根节点分别为 leftChild_i 和 rightChild_i ，若左子

树包含 k 个节点 (其中 $0 \leq k \leq j-1$) , 最大权值为 $f(\text{leftChild}_i, k)$, 则右子树包含 $j-1-k$ 个节点, 最大权值为 $f(\text{rightChild}_i, j-1-k)$ 。因此选取所有 k 的选择中最大的权值即可, 即 $f(i, j) = \max\{f(\text{leftChild}_i, k) + f(\text{rightChild}_i, j-1-k) + v_i\}$;

最终在 $f(i, m)$ 中选择权值最大的作为最终的最大权值 (其中 $i \in [0, n)$)。该算法的时间复杂度是 $O(n^2)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

MultipleTreeDP 多叉树动规

- [Multiple Tree DP - 多叉树动规](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Multiple Tree DP - 多叉树动规

问题

与<Binary Tree DP>类似，对于拥有 n 个节点的多叉树，其节点下标范围为 $[0, n)$ ，节点 i 的权值为 v_i ($v_i > 0$)，整个多叉树的权值为所有节点的权值之和。现在要求只保留 m 个节点 ($0 < m < n$)，剪裁掉 $n-m$ 个节点，要求剩余部分仍然是一个多叉树，而不能是多个树。

对于拥有 n 个节点的多叉树，求出保留 m 个节点的多叉树的最大权值。

解法

与<Binary Tree DP>思路类似，仍然设 $f(i, j)$ 表示以节点 i 为根节点的树上，保留 j 个节点（包括节点 i 自己）的最大权值。其转移方程如下：

$$f(i, j) = \begin{cases} v_i & \text{(初始化)} \\ \max \{ \sum_{1}^j f(\text{child}_j, k_j) + v_i \} & \text{if } i, j, k \in [0, n), i \neq j, \sum_{1}^j k_j = m-1 \end{cases}$$

(1) 节点数量为 1 的二叉树，其最大权值即为节点自己的权值，即 $f(i, i) = v_i$ ；

(2) 对于以 i 为根节点的多叉树，假设它拥有 j 个子树，每个子树的根节点分别为 child_j 。子树 j 保留 k_j 个节点，那么所有子树的节点之和即为 $\sum_{1}^j k_j = m-1$ （加上根节点 i 自己一共 m 个节点）。因此在所有可能中选取最大的权值之和即可；

最终在 $f(i, m)$ 中选择权值最大的作为最终的最大权值（其中 $i \in [0, n)$ ）。该算法的时间复杂度是 $O(n^2)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



MultipleTreeDPExtension 多叉树动规扩展

- [Multiple Tree DP Extension - 多叉树动规扩展](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Multiple Tree DP Extension - 多叉树动规扩展

问题

与<Binary Tree DP>类似，对于拥有 n 个节点的多叉树，其节点下标范围为 $[0, n)$ ，节点 i 的权值为 v_i ($v_i > 0$)，整个多叉树的权值为所有节点的权值之和。现在要求只保留 m 个节点 ($0 < m < n$)，剪裁掉 $n-m$ 个节点，要求剩余部分仍然是一个多叉树，而不能是多个树。

对于拥有 n 个节点的多叉树，求出保留 m 个节点的多叉树的最大权值。

解法

与<Binary Tree DP>思路类似，仍然设 $f(i, j)$ 表示以节点 i 为根节点的树上，保留 j 个节点（包括节点 i 自己）的最大权值。其转移方程如下：

$$\begin{aligned}
 f(i, j) = & \\
 & \begin{cases}
 v_i & \text{(初始化)} \\
 i, j \in [0, n), i = j \\
 \max \{ \sum_{1}^j f(\text{child}_j, k_j) + v_i \} & i, j, k \in [0, n), i \neq j, \sum_{1}^j k_j = m-1
 \end{cases} \\
 & \end{cases}
 \end{aligned}$$

(1) 节点数量为 1 的二叉树，其最大权值即为节点自己的权值，即 $f(i, i) = v_i$ ；

(2) 对于以 i 为根节点的多叉树，假设它拥有 j 个子树，每个子树的根节点分别为 child_j 。子树 j 保留 k_j 个节点，那么所有子树的节点之和即为 $\sum_{1}^j k_j = m-1$ （加上根节点 i 自己一共 m 个节点）。因此在所有可能中选取最大的权值之和即可；

最终在 $f(i, m)$ 中选择权值最大的作为最终的最大权值（其中 $i \in [0, n)$ ）。该算法的时间复杂度是 $O(n^2)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



LoopedMultipleTreeDP 带环多叉树动规

- [LoopedMultipleTreeDP 带环多叉树动规](#)

LoopedMultipleTreeDP 带环多叉树动规

TraverseBinaryTreeDP 遍历二叉树动规

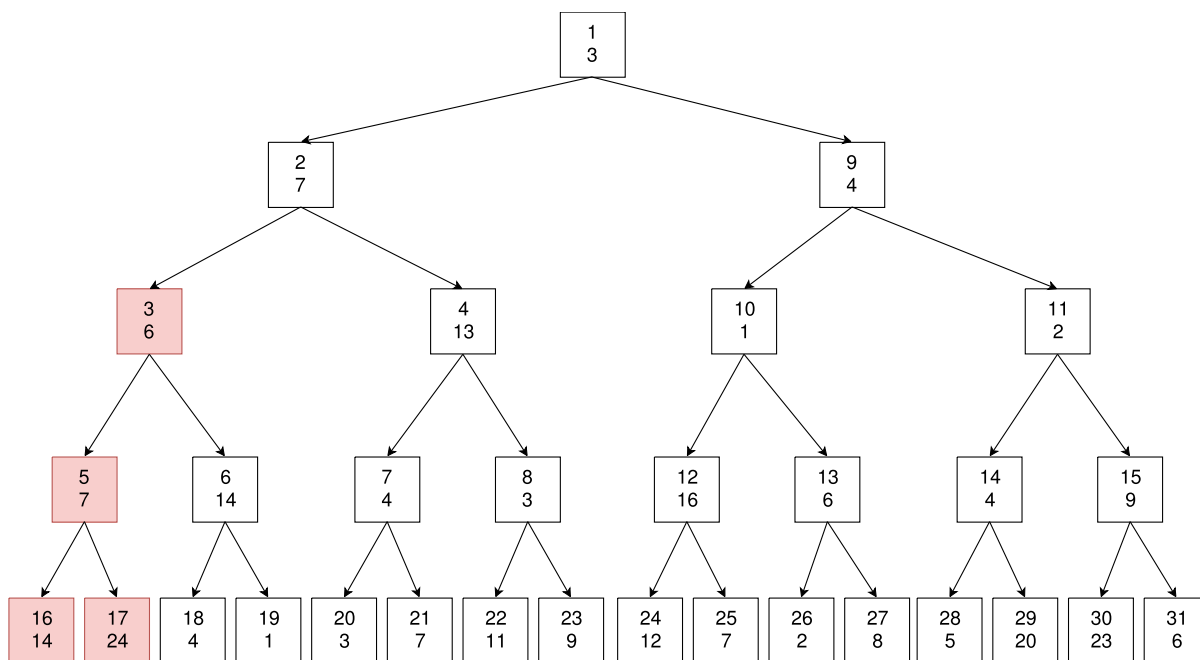
- [Traverse Binary Tree DP - 遍历二叉树动规](#)
 - [问题](#)
 - [解法](#)
 - [USACO Mar 2008 "Cow Travelling" \(游荡的奶牛\)](#)
 - [源码](#)
 - [测试](#)

Traverse Binary Tree DP - 遍历二叉树动规

问题

在一个二叉树上，从任意节点 i 到达另一个任意节点 j 的路线是唯一的。假设该二叉树上的每个节点都是一个牧场，而每个牧场中都有一只奶牛，节点 i 的奶牛拥有一个权值，为 v_i ，奶牛会在二叉树上游荡，但它游荡的位置不超过一个距离，为 $Dist$ 。节点 i 的牧场上拥有的奶牛数量并不固定，可能拥有 0 只奶牛，那么节点 i 拥有的权值为 0 ；可能拥有 n 只奶牛，那么节点 i 拥有的权值为这 n 只奶牛的权值之和，即 $\sum_{k=1}^n v_k$ （其中 $k \in [1, n]$ ）。求出所有节点中权值最大的节点的权值。

对于下图中的二叉树，每个节点的标号为上面的数字，权值为下面的数字。但奶牛游荡的距离为 1 时，会拥有最大权值的节点为节点 5 ，最大权值为 $51 = 6+7+14+24$ ，即节点 3 、 5 、 16 、 17 的权值之和：

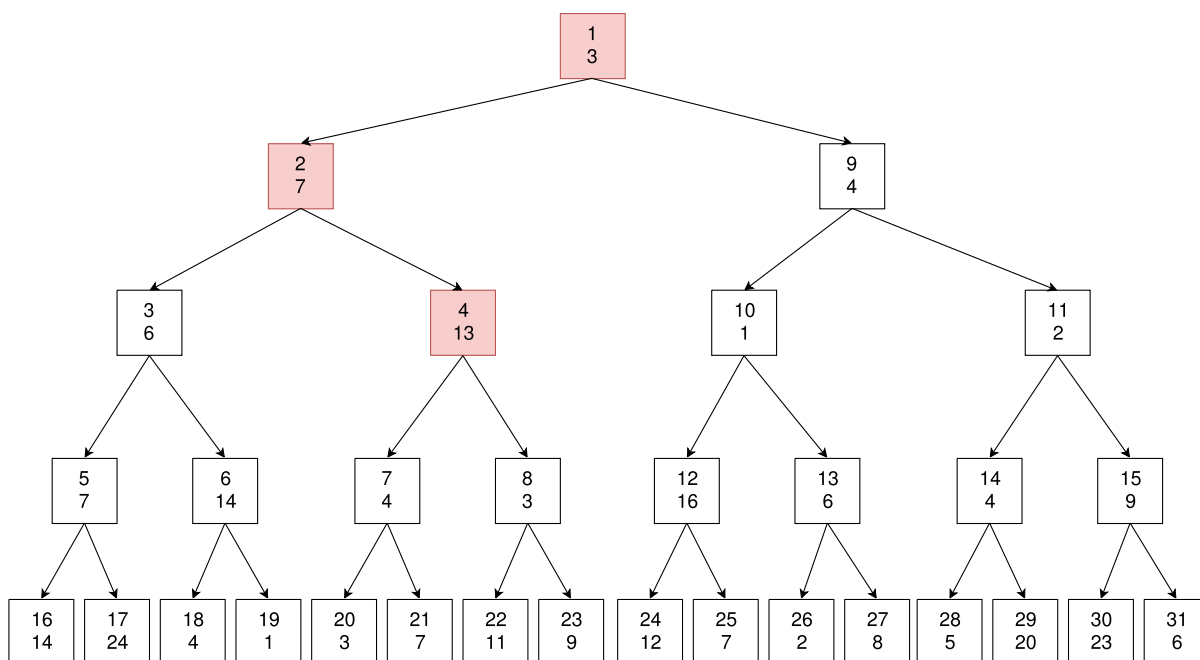


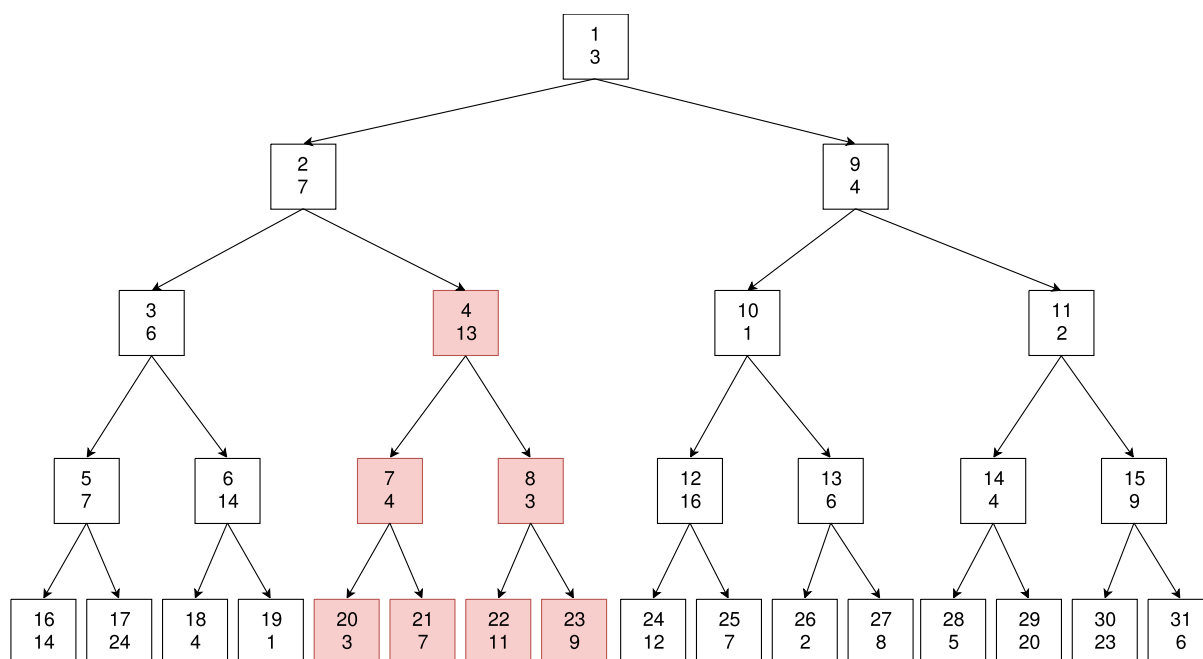
本问题的原型为USACO Mar 2008 “Cow Travelling”（游荡的奶牛）。

解法

问题中的示例是一种最简单的情况，即游荡距离为 1，这时节点 i 的最大权值即为该节点与所有相邻节点的权值之和。当游荡距离增大，节点 i 的最大权值为所有到该节点的距离不超过 Dist 的节点的权值之和，即 $\sum_{1 \leq k \leq n} v_k$ （其中 n 为所有节点的数量， k 节点的权值为 v_k ）。

节点 i 可以到达二叉树的向上和向下 Dist 层的所有节点，如图：





对于节点 4 的奶牛，当其 $\text{Dist} = 2$ ，则向上可以到达节点 2、1；向下可以到达 7、8、20、21、22、23。

(1) 向上可达的所有节点的权值和：

设 $\text{up}(i, j)$ 为游荡距离为 j 的节点 i ，向上可以到达的所有节点的权值之和，则有 $\text{up}(i, j) = \text{up}(\text{father}_i, j-1) + v\{\text{father}_i\}$ ，即游荡距离为 j 的节点 i ，其向上可达的权值和，等于游荡距离为 $j-1$ 的父节点 father_i 的向上可达的权值和与父节点自己的权值之和。在上图中可以看出，游荡距离为 2 的节点 4，其向上权值和 $\text{up}(4, 2)$ ，恰好等于游荡距离为 1 的节点 2 的向上权值和与节点 2 的权值之和，即 $\text{up}(4, 2) = \text{up}(2, 1) + v_2$ 。

对于游荡距离为 0 的节点 i ，其向上权值和为 $\text{up}(i, 0) = 0$ 。

(2) 向下可达的所有节点的权值和：

设 $\text{down}(i, j)$ 为游荡距离为 j 的节点 i ，向下可以到达的所有节点的权值之和，则有 $\text{down}(i, j) = \text{down}(\text{leftChild}_i, j-1) + \text{down}(\text{rightChild}_i, j-1) + v\{\text{leftChild}_i\} + v\{\text{rightChild}_i\}$ ，即游荡距离为 j 的节点 i ，其向下可达的权值和，等于游荡距离为 $j-1$ 的左右孩子节点 leftChild_i 和 rightChild_i 的向下可达权值和，与左右孩子节点自己的权值的总和。在上图中可以看出，游荡距离为 2 的节点 4，其向下权值和 $\text{down}(4, 2)$ ，恰好等于游荡距离为 1 的节点 7、8 的向下权值和，与节点 7、8 的权值的总和，即 $\text{down}(4, 2) = \text{down}(7, 1) + \text{down}(8, 1) + v_7 + v_8$ 。

对于游荡距离为 0 的节点 i ，其向下权值和为 $\text{down}(i, 0) = 0$ 。

根据 (1) 和 (2) 两个部分，可以得出游荡距离为 j 的节点 i 的最大权值 $f(i, j) = \text{up}(i, j) + \text{down}(i, j) + v_i$ 。

最终在所有 $f(i, j)$ 中选择最大值作为返回结果（其中 $i, j \in [0, n)$ ）。该算法的时间复杂度是 $O(n^2)$ 。

USACO Mar 2008 “Cow Travelling”（游荡的奶牛）

- <http://train.usaco.org/TESTDATA/MAR08.ctravel.htm>
-

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

Chapter-5 GraphTheory 第5章 图论

- [Chapter-5 Graph Theory](#)
- [第5章 图论](#)

Chapter-5 Graph Theory

第5章 图论

1. Traverse - 遍历
 - i. [KnowledgePoint](#) - 知识要点
 - ii. [PreorderTraverse](#) - 先序遍历
 - iii. [InorderTraverse](#) - 中序遍历
 - iv. [PostorderTraverse](#) - 后序遍历
 - v. [LevelorderTraverse](#) - 层序遍历
 - vi. [DepthFirstSearch\(DFS\)](#) - 深度优先搜索
 - vii. [BreadthFirstSearch\(BFS\)](#) - 广度优先搜索
 - viii. [TopologicalSort](#) - 拓扑排序
 - ix. [EulerCycle](#) - 欧拉回路
2. MinimumSpanningTree - 最小生成树
 - i. [KnowledgePoint](#) - 知识要点
 - ii. [Kruskal](#) - Kruskal算法
 - iii. [Prim](#) - Prim算法
 - iv. [SecondMinimumSpanningTree](#) - 次小生成树
 - v. [OptimalRatioSpanningTree](#) - 最优比率生成树
3. ShortestPath - 最短路径
 - i. [KnowledgePoint](#) - 知识要点
 - ii. [Relaxation](#) - 松弛操作
 - iii. [BellmanFord](#) - BellmanFord算法
 - iv. [ShortestPathFasterAlgorithm](#) - 最短路径更快算法 (SPFA)
 - v. [Dijkstra](#) - Dijkstra算法
 - vi. [Floyd](#) - Floyd算法
 - vii. [DifferentConstraints](#) - 差分约束
4. Connectivity - 连通
 - i. [KnowledgePoint](#) - 知识要点
 - ii. [Kosaraju](#) - Kosaraju算法
 - iii. [Tarjan](#) - Tarjan算法
 - iv. [Gabow](#) - Gabow算法

- v. [TwoSatisfiability](#) - 2-SAT问题
- vi. [Cut](#) - 割
- vii. [DoubleConnectedComponent](#) - 双联通分支
- viii. [LeastCommonAncestor](#) - 最近公共祖先
- ix. [RangeExtremumQuery](#) - 区域最值查询
- 5. [FlowNetwork](#) - 网络流
 - i. [EdmondsKarp](#) - EdmondsKarp算法
 - ii. [PushAndRelabel](#) - 压入与重标记
 - iii. [Dinic](#) - Dinic算法
 - iv. [DistanceLabel](#) - 距离标号算法
 - v. [RelabelToFront](#) - 重标记与前移算法
 - vi. [HighestLabelPreflowPush](#) - 最高标号预留与推进算法
 - vii. [DistanceLabel_AdjacentListVersion](#) - 距离标号算法-邻接表优化版
 - viii. [Summary-Maxflow](#) - 最大流算法小结
 - ix. [MinimumCost_Maxflow](#) - 最小费用最大流
 - x. [MultipleSourceMultipleSink_Maxflow](#) - 多源点、多汇点最大流
 - xi. [Connectivity](#) - 连通度
 - xii. [NoSourceNoSink_VolumeBounded_Flow](#) - 无源点、无汇点、容量有上下界的流网络
 - xiii. [VolumeBounded_Maxflow](#) - 容量有上下界的最大流
 - xiv. [VolumeBounded_Minflow](#) - 容量有上下界的最小流
- 6. [BinaryMatch](#) - 二分匹配
 - i. [Hungarian](#) - 匈牙利算法
 - ii. [HopcroftKarp](#) - Hopcroft-Karp算法
 - iii. [MatchToMaxflow](#) - 二分匹配转化为最大流
 - iv. [KuhnMunkres](#) - Kuhn-Munkres算法
 - v. [Introduction-Domination_Independent_Covering_Clique](#) - 支配集、独立集、覆盖集、团的介绍
 - vi. [WeightedCoveringAndIndependentSet](#) - 最小点权覆盖和最大点权独立集
 - vii. [MinimumDisjointPathCovering](#) - 最小不相交路径覆盖
 - viii. [MinimumJointPathCovering](#) - 最小可相交路径覆盖
 - ix. [Coloring](#) - 染色问题

Section-1 Traverse 第1节 遍历

Section-1 Traverse 第1节 遍历

- [KnowledgePoint](#) 知识要点
- [PreorderTraverse](#) 先序遍历
- [InorderTraverse](#) 中序遍历
- [PostorderTraverse](#) 后序遍历
- [LevelorderTraverse](#) 层序遍历
- [DepthFirstSearch](#) 深度优先搜索
- [BreadthFirstSearch](#) 广度优先搜索
- [TopologicalSort](#) 拓扑排序
- [EulerCycle](#) 欧拉回路

KnowledgePoint 知识要点

- Knowledge Point - 知识要点
 - 图
 - 子图 (Subgraph)
 - 完全图 (Complete Graph)
 - 无向边
 - 有向边
 - 出度
 - 入度
 - 度数
 - 环
 - 拓扑排序
 - 欧拉路径 (Eulerian Path)
 - 欧拉回路 (Eulerian Cycle)
 - 欧拉图 (Eulerian Diagram)
 - 汉密尔顿路径 (Hamilton Path)
 - 汉密尔顿回路 (Hamilton Cycle)
 - 汉密尔顿图 (Hamilton Diagram)
 - 图论术语

Knowledge Point - 知识要点

图

图 $G = \langle V, E \rangle$ 是由顶点集合 V 和边集合 E 组成的数据结构。一个边为连接两个顶点的曲线，若两个顶点 u 和 v 为一条边的两个端点，则称 u 和 v 相邻。

子图 (Subgraph)

一个所有顶点和边都属于图 G 的图，称为 G 的子图。

完全图 (Complete Graph)

所有顶点两两相邻的图称为完全图。

无向边

若无向边 e 的两端点是 u 和 v ，则可以从 u 出发到达 v ，也可以从 v 出发到达 u 。

有向边

若有向边 e 从 u 指向 v ，则只能从 u 出发到达 v ，而不能反向。无向边也可以看作相同两个端点之间的两条反向有向边的叠加。

出度

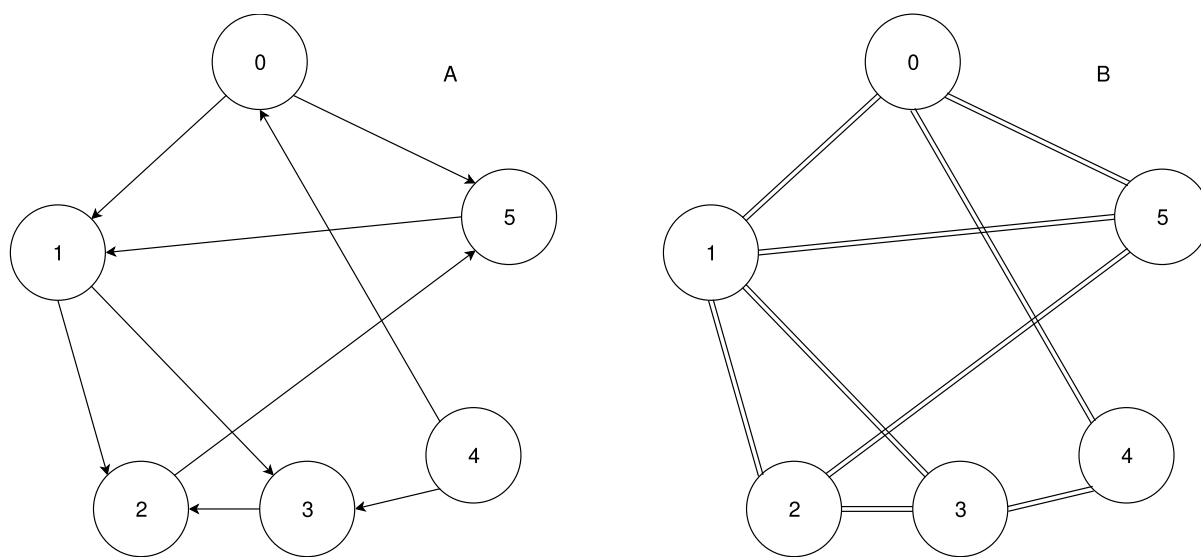
节点 u 的出度是从节点 u 出发的边的数量，也称为出度度数，从节点 u 出发的边也称为出弧边。对于无向边和无向图来说，节点 u 的所有边都可以看作出弧边，即边数等于出度。

入度

节点 u 的入度是到达节点 u 的边的数量，也称为入度度数，到达节点 u 的边也称为入弧边。对于无向边和无向图来说，节点 u 的所有边也都可以看作入弧边，即边数等于入度。无向图中每个节点的出度和入度相等。

度数

图 G 中的节点 u 所关联的边数，称作该节点的度数。无向图的任意节点 v_i 的度数等于出度度数，也等于入度度数。有向图的任意节点 v_i 的度数等于出度度数与入度度数之和。



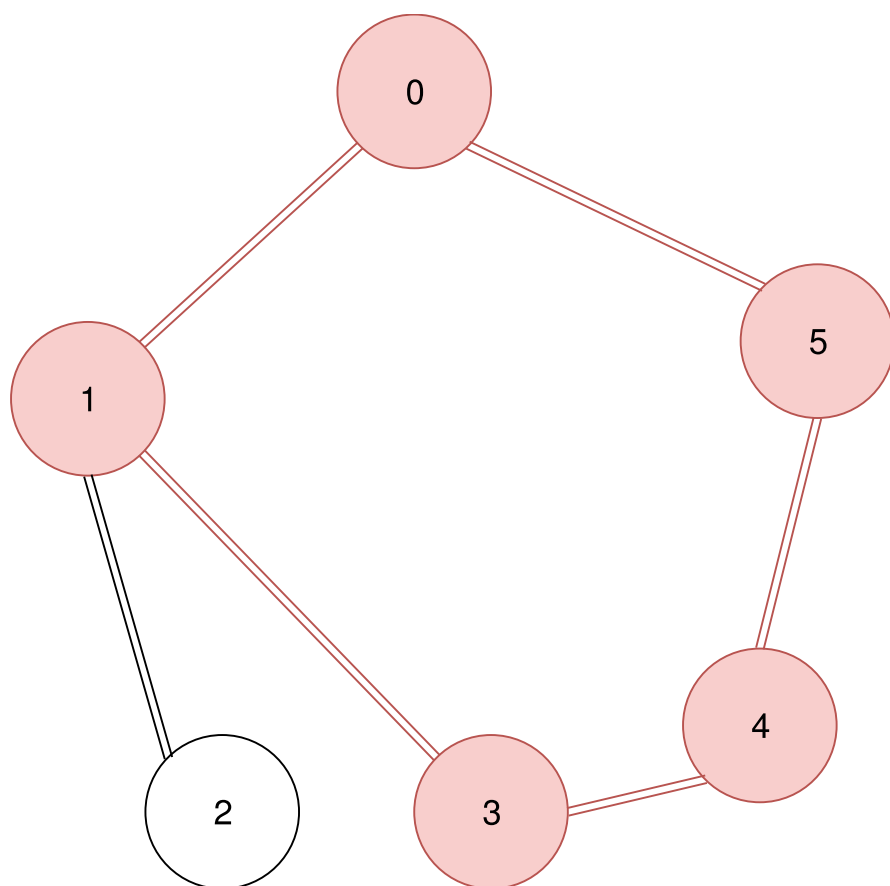
上面两个图中，图 A 为有向图，节点 0 - 6 的出度分别为 2, 2, 1, 1, 2, 1，入度分别为 1, 1, 2, 2, 0, 2。图 B 为无向图，节点 0 - 6 的出度分别为 3, 4, 3, 3, 2, 3，入度与出度一样。

$n \times n$ 的矩阵 g 可以表示一个拥有 n 个节点的图，节点下标范围为 $[1, n]$ 。
 $g[i, j]$ 表示从节点 i 到 j 的距离 ($i, j \in [1, n]$)，也可以是其他信息，比如节点 i 是否可以直接到达节点 j 等等。无向图中相连的两个节点，可以看作有向图中两个节点之间有权值相等，方向相反的两条边。图 A 和 B 可以表示为：

$$\begin{aligned}
 &A = \\
 &\quad \begin{matrix} \backslash \text{begin}\{\text{bmatrix}\} \\
 0\{0,0\} \ \& \ 1\{0,1\} \ \& \ 0\{0,2\} \ \& \ 0\{0,3\} \ \& \ 0\{0,4\} \ \& \ 1\{0,5\} \ \backslash \\
 0\{1,0\} \ \& \ 0\{1,1\} \ \& \ 1\{1,2\} \ \& \ 1\{1,3\} \ \& \ 0\{1,4\} \ \& \ 0\{1,5\} \ \backslash \\
 0\{2,0\} \ \& \ 0\{2,1\} \ \& \ 0\{2,2\} \ \& \ 0\{2,3\} \ \& \ 0\{2,4\} \ \& \ 1\{2,5\} \ \backslash \\
 0\{3,0\} \ \& \ 0\{3,1\} \ \& \ 1\{3,2\} \ \& \ 0\{3,3\} \ \& \ 0\{3,4\} \ \& \ 0\{3,5\} \ \backslash \\
 1\{4,0\} \ \& \ 0\{4,1\} \ \& \ 0\{4,2\} \ \& \ 1\{4,3\} \ \& \ 0\{4,4\} \ \& \ 0\{4,5\} \ \backslash \\
 0\{5,0\} \ \& \ 1\{5,1\} \ \& \ 0\{5,2\} \ \& \ 0\{5,3\} \ \& \ 0\{5,4\} \ \& \ 0\{5,5\} \\
 \backslash \text{end}\{\text{bmatrix}\} \end{matrix} \\
 &\quad \quad \quad \backslash \text{quad} \\
 &B = \\
 &\quad \begin{matrix} \backslash \text{begin}\{\text{bmatrix}\} \\
 0\{0,0\} \ \& \ 1\{0,1\} \ \& \ 0\{0,2\} \ \& \ 0\{0,3\} \ \& \ 1\{0,4\} \ \& \ 1\{0,5\} \ \backslash \\
 1\{1,0\} \ \& \ 0\{1,1\} \ \& \ 1\{1,2\} \ \& \ 1\{1,3\} \ \& \ 0\{1,4\} \ \& \ 0\{1,5\} \ \backslash \\
 0\{2,0\} \ \& \ 1\{2,1\} \ \& \ 0\{2,2\} \ \& \ 1\{2,3\} \ \& \ 0\{2,4\} \ \& \ 1\{2,5\} \ \backslash \\
 0\{3,0\} \ \& \ 1\{3,1\} \ \& \ 1\{3,2\} \ \& \ 0\{3,3\} \ \& \ 1\{3,4\} \ \& \ 0\{3,5\} \ \backslash \\
 1\{4,0\} \ \& \ 0\{4,1\} \ \& \ 0\{4,2\} \ \& \ 1\{4,3\} \ \& \ 0\{4,4\} \ \& \ 0\{4,5\} \ \backslash \\
 1\{5,0\} \ \& \ 1\{5,1\} \ \& \ 1\{5,2\} \ \& \ 0\{5,3\} \ \& \ 0\{5,4\} \ \& \ 0\{5,5\} \\
 \backslash \text{end}\{\text{bmatrix}\} \end{matrix}
 \end{aligned}$$

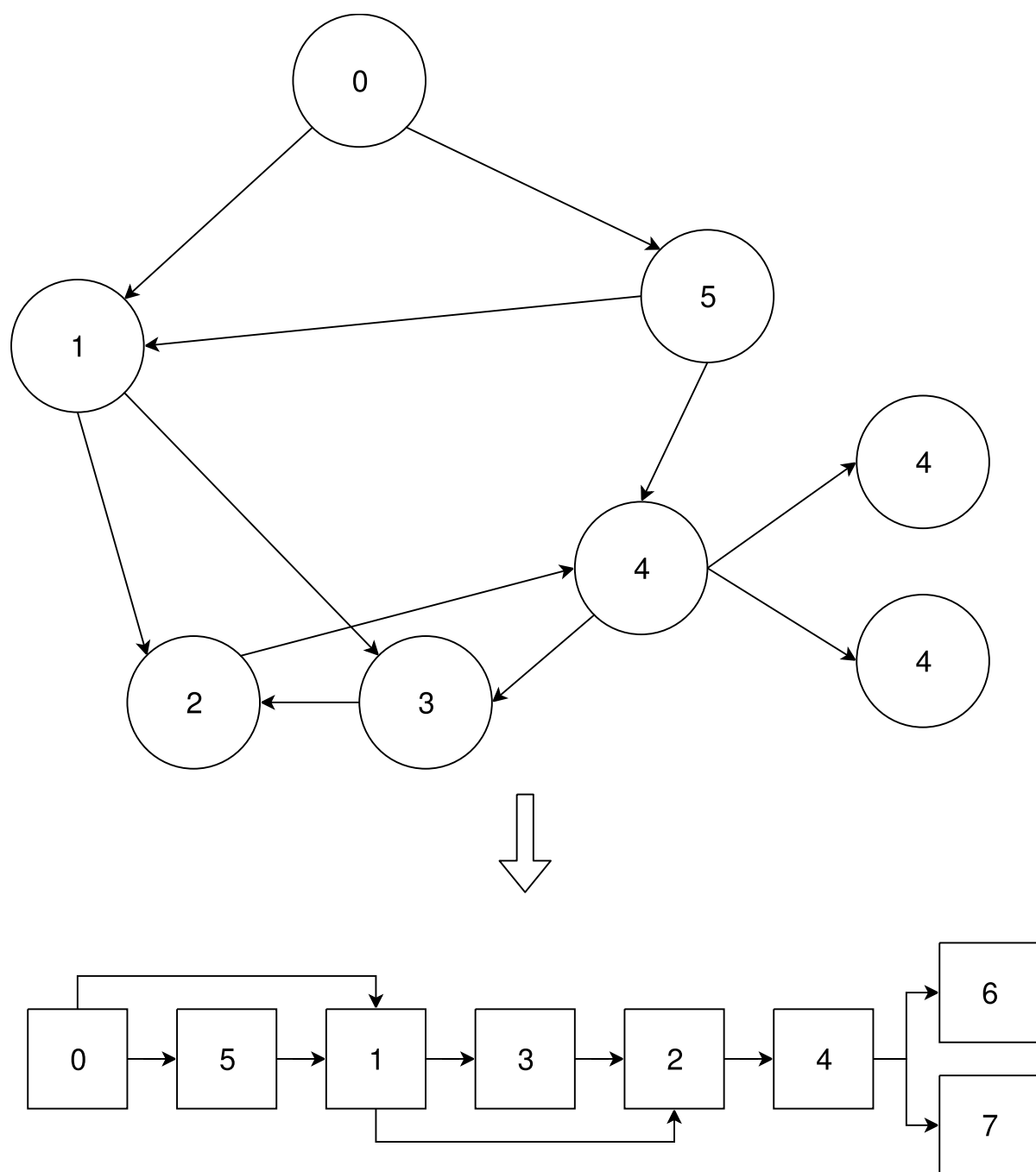
环

若图 G 中存在一个节点 i ，从它出发可以返回自己，则称这条路径为一个环。不存在环的图称为无环图。下图是一个五向有环图：



拓扑排序

不存在环的有向图中存在拓扑排序。在可以拓扑排序的有向图 G 中，节点可以分为三类：起点，终点和中间点。起点只有出弧边，没有入弧边；终点只有入弧边，没有出弧边；中间点既有出弧边也有入弧边。下图是一个有向无环图的拓扑排序：



欧拉路径 (Eulerian Path)

图 G 中存在这样的一条路径，经过每条边一次且仅一次（同一个顶点可以经过多次），可以遍历图中的所有边，则这条路径称为欧拉路径。有向图 DG 中存在一个起始顶点 v_1 满足 $degree\{out\} = degree\{in\} + 1$ （出度比入度大1），存在一个终止顶点 v_2 满足 $degree\{in\} = degree\{out\} + 1$ （入度比出度大1），其余所有顶点的入度等于出度，则该有向图 DG 中存在欧拉路径。无向图 UG 中存在两个顶点 v_1 和 v_2 满足度数为奇数，其余节点的度数都是偶数，则该无向图 UG 中存在欧拉路径。

欧拉回路 (Eulerian Cycle)

若图 G 中存在欧拉路径，且该路径为一个回路，则称该路径为欧拉回路。有向图 DG 的任意顶点 v_i 满足 $degree\{in\} = degree\{out\}$ ，出度等于入度，则该有向图 DG 中存在欧拉回路。无向图 UG 的任意顶点 v_i 满足读数为偶数，则该无向图 DG 中存在欧拉回路。

欧拉图 (Eulerian Diagram)

拥有欧拉回路的图 G 称为欧拉图。

汉密尔顿路径 (Hamilton Path)

图 G 中存在这样的一条路径，经过每个顶点一次且仅一次（同一条边可以经过多次），可以遍历图中的所有顶点，则这条路径称为汉密尔顿路径。求解汉密尔顿路径是一个NP完全问题。

汉密尔顿回路 (Hamilton Cycle)

图 G 中存在汉密尔顿路径，且该路径为一个回路，则称该路径为汉密尔顿回路。

汉密尔顿图 (Hamilton Diagram)

拥有汉密尔顿回路的图 G 称为汉密尔顿图。完全图必然是汉密尔顿图。

图论术语

- <https://zh.wikipedia.org/wiki/%E5%9B%BE%E8%AE%BA%E6%9C%AF%E8%AF%AD>



PreorderTraverse 先序遍历

- [Preorder Traverse - 先序遍历](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

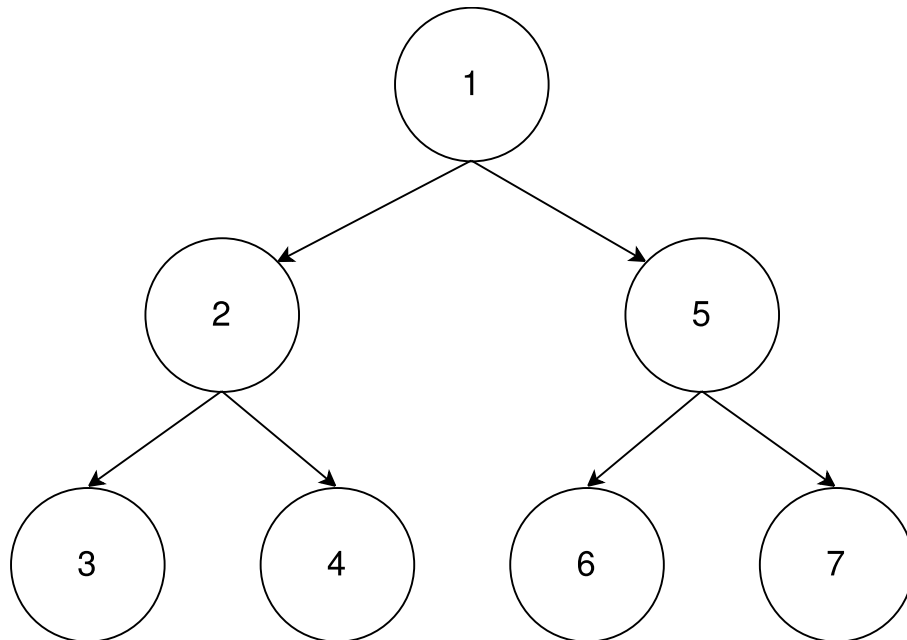
Preorder Traverse - 先序遍历

问题

用先序遍历的方式来遍历二叉树。

解法

从二叉树根节点 `root` 开始，递归的对二叉树上的每个节点 `i`，总是优先访问节点 `i` 本身，然后访问 `i` 的左孩子节点，最后访问 `i` 的右孩子节点。如图：



先序遍历的时间复杂度是 $O(n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

InorderTraverse 中序遍历

- [Inorder Traverse - 中序遍历](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

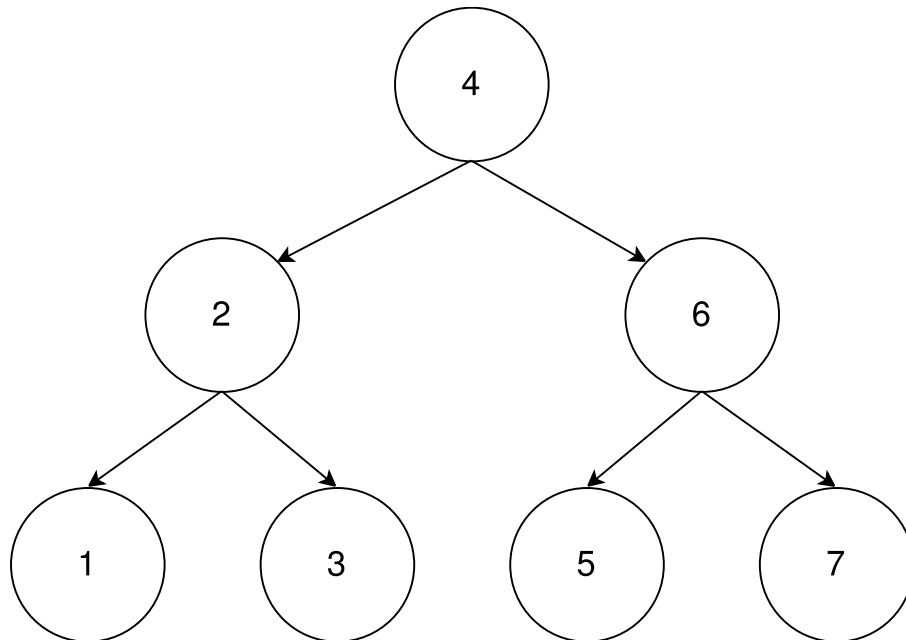
Inorder Traverse - 中序遍历

问题

用中序遍历的方式来遍历二叉树。

解法

从二叉树根节点 `root` 开始，递归的对二叉树上的每个节点 `i`，总是优先访问节点 `i` 的左孩子节点，然后访问 `i` 节点本身，最后访问 `i` 的右孩子节点。如图：



中序遍历的时间复杂度是 $O(n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

PostorderTraverse 后序遍历

- [Postorder Traverse - 后序遍历](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

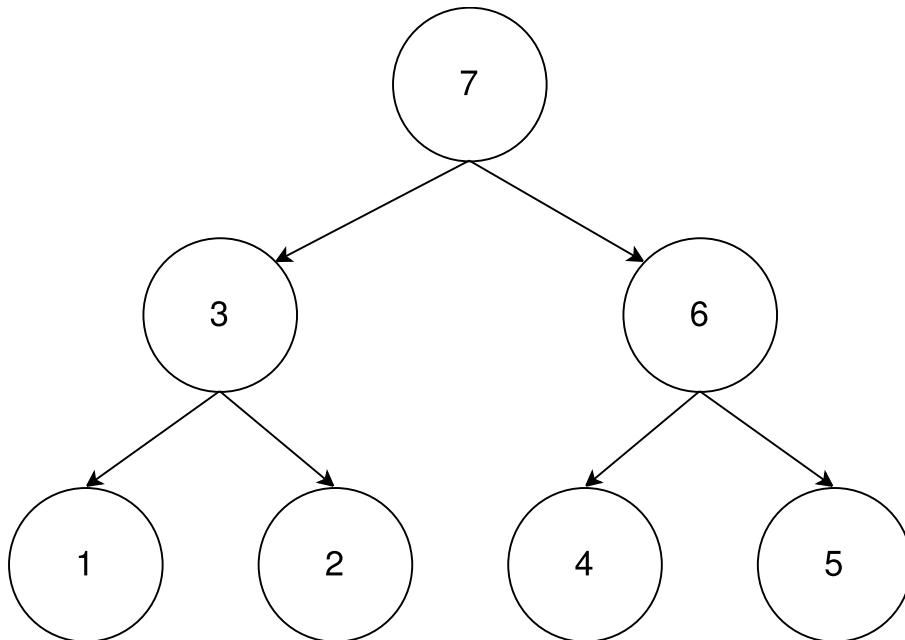
Postorder Traverse - 后序遍历

问题

用后序遍历的方式来遍历二叉树。

解法

从二叉树根节点 `root` 开始，递归的对二叉树上的每个节点 `i`，总是优先访问节点 `i` 的左孩子节点，然后访问 `i` 的右孩子节点，最后访问 `i` 节点本身。如图：



后序遍历的时间复杂度是 $O(n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

LevelorderTraverse 层序遍历

- [Levelorder Traverse - 层序遍历](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

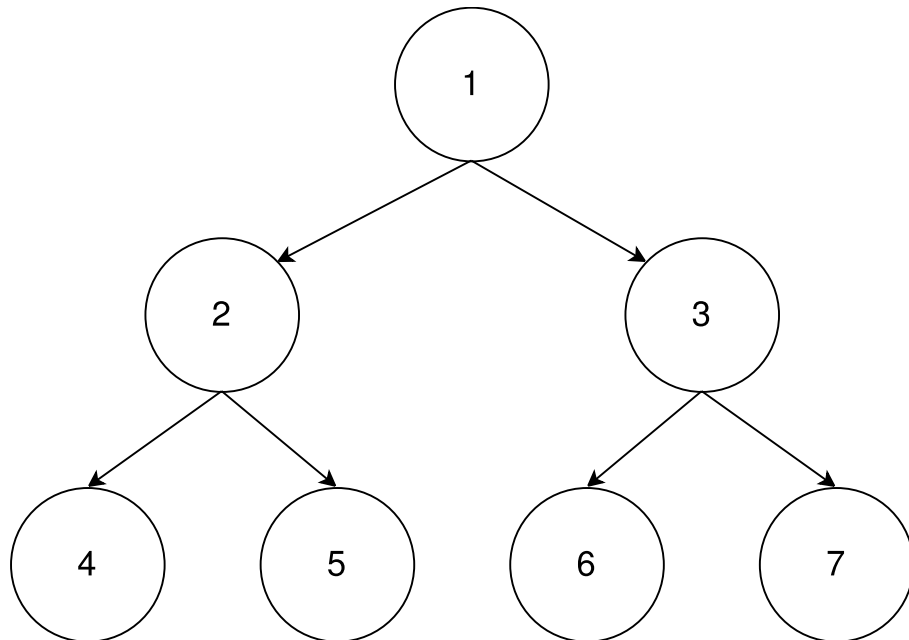
Levelorder Traverse - 层序遍历

问题

用层序遍历的方式来遍历二叉树。

解法

从二叉树根节点 `root` 开始，递归的对二叉树上的每个节点 `i`，总是优先访问节点 `i` 以及与其处于同一高度的节点，然后再访问 `i` 以及与其处于同一高度的节点的孩子节点们。如图：



先序遍历、中序遍历和后序遍历都可以比较容易的用递归来实现，而层序遍历是无法用递归函数来实现的。我们可以借助队列来实现层序遍历。初始时将二叉树的根节点 `root` 放入队列中，之后每次从队列中取出一个节点进行访问，并将该节点的左右孩子节点放入队列，直到队列为空，算法结束。在这个过程中，队列对所有节点的访问顺序进行控制，在上图中，保证对于节点 `1`，总是先访问它的孩子节点 `2` 和 `3`，然后再访问 `2` 和 `3` 各自的孩子节点 `4, 5, 6, 7`。

层序遍历的时间复杂度是 $O(n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```


DepthFirstSearch 深度优先搜索

- [Depth First Search\(DFS\) - 深度优先搜索](#)
 - [问题](#)
 - [解法](#)
 - [深度优先搜索](#)
 - [源码](#)
 - [测试](#)

Depth First Search(DFS) - 深度优先搜索

问题

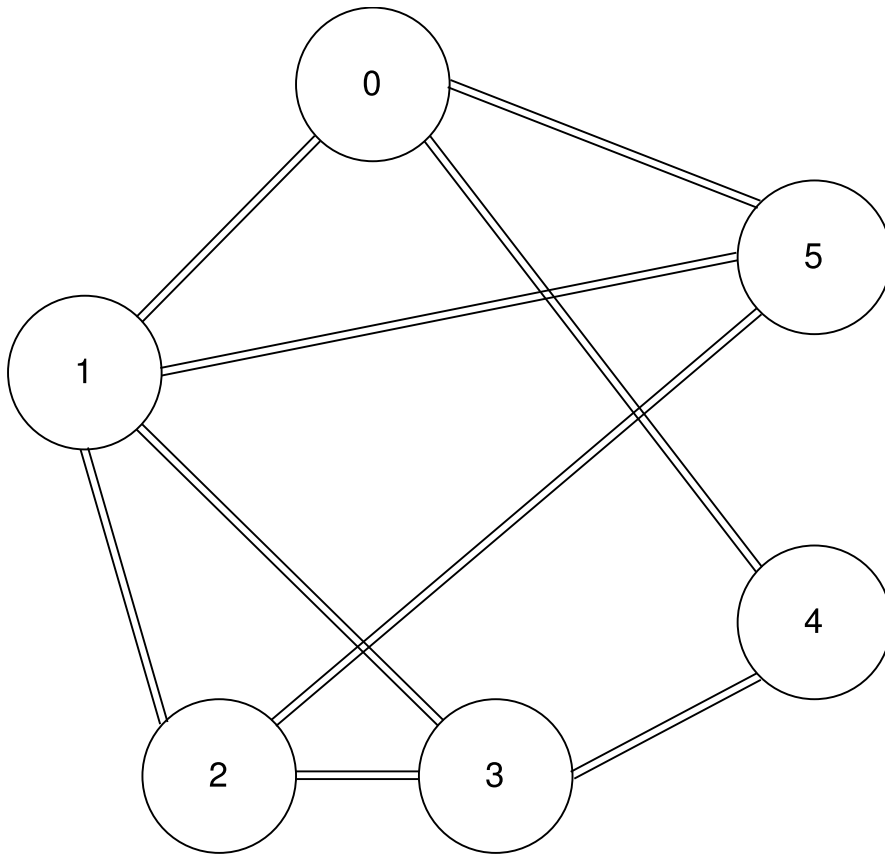
用深度优先搜索从图 G 的节点 beg 开始，遍历图 G 中的所有节点。

解法

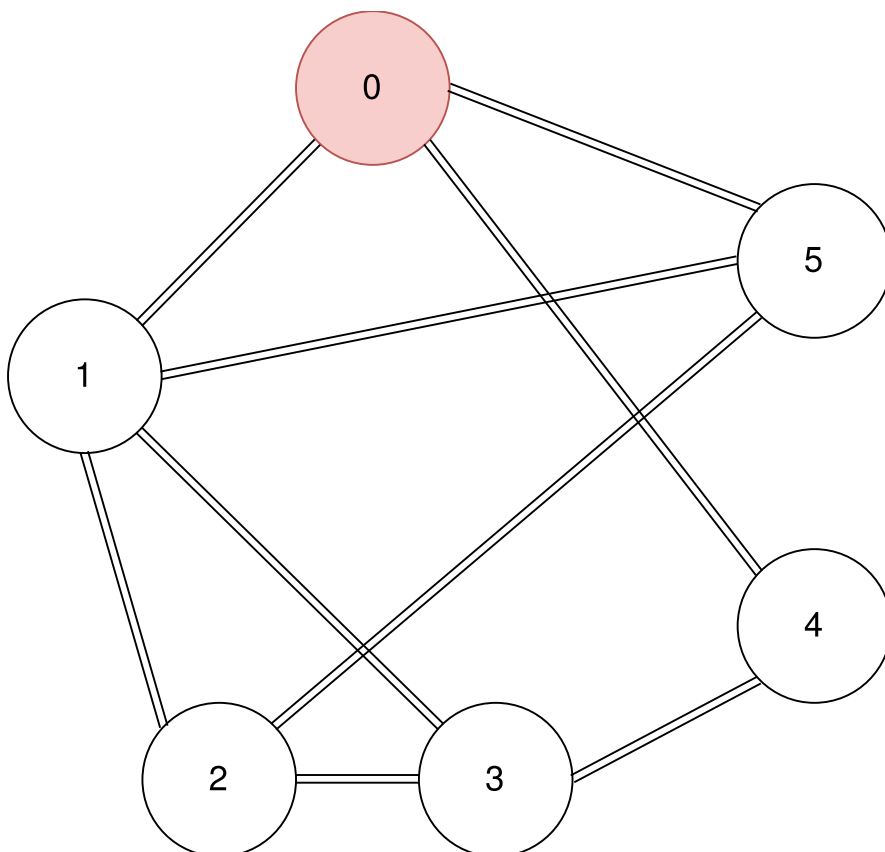
在图 G 中，假设节点 i 的邻节点集合为 V_i ，类似于二叉树的先序遍历，对于图中的任意节点 i ，在访问节点 i 之后，从该节点的邻节点集合 V_i 中挑选其中一个 j ，继续递归的重复该遍历操作，直到没有更加深入的节点可以搜索时，再返回上一层，考虑邻节点集合 V_i 中的下一个节点。

从某节点 V_i 开始DFS，遍历结束后会得到一串节点，即为从 V_i 开始遍历节点的顺序。我们称这串节点的最后一个节点为图 G 中以 V_i 为起点的 DFS 终点。这串节点的数量即为 V_i 到终点的搜索距离 $Distance$ ，也可以称为DFS的搜索时间，即节点 V_i 进行一次DFS所需的时间（将遍历一个节点的时间看作单位时间）。很多算法中都会利用搜索距离，而不太关心搜索到的这串节点的具体内容。

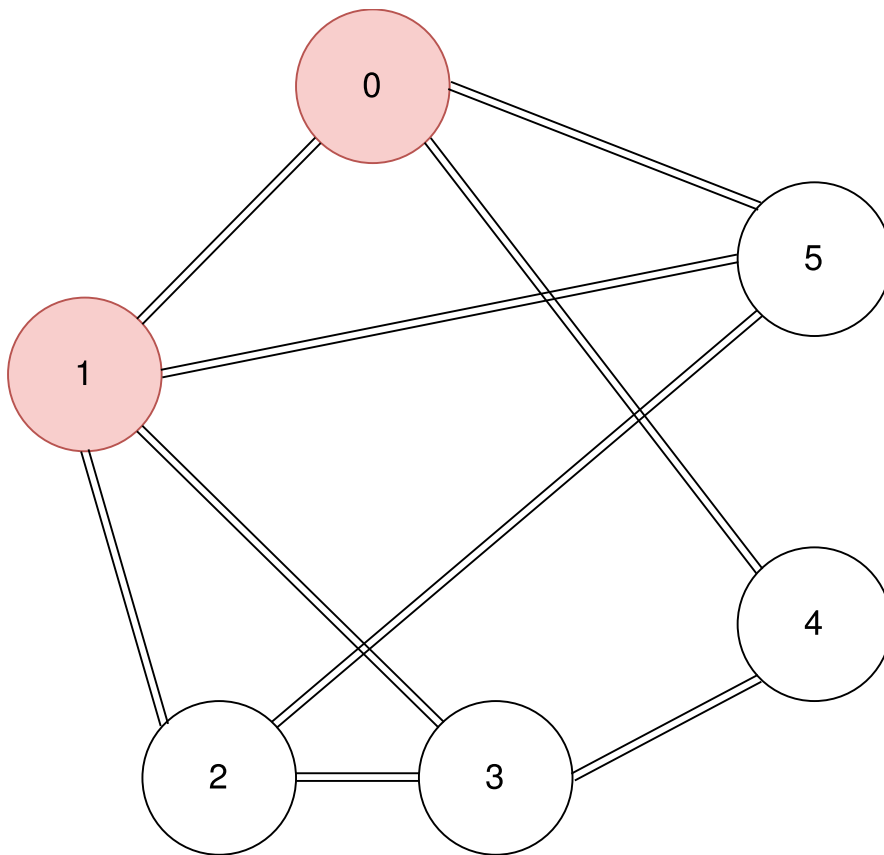
在整个遍历过程中，为了避免重复的访问一个节点，在访问了某个节点 i 之后，我们将它染成红色（实际编码中，可以设置一个数组 `visited`，通过 `visited_i = true \mid false` 来标记某个节点 i 时候被访问过）。下面演示从无向图 G 中的节点 0 开始进行深度优先搜索过程：



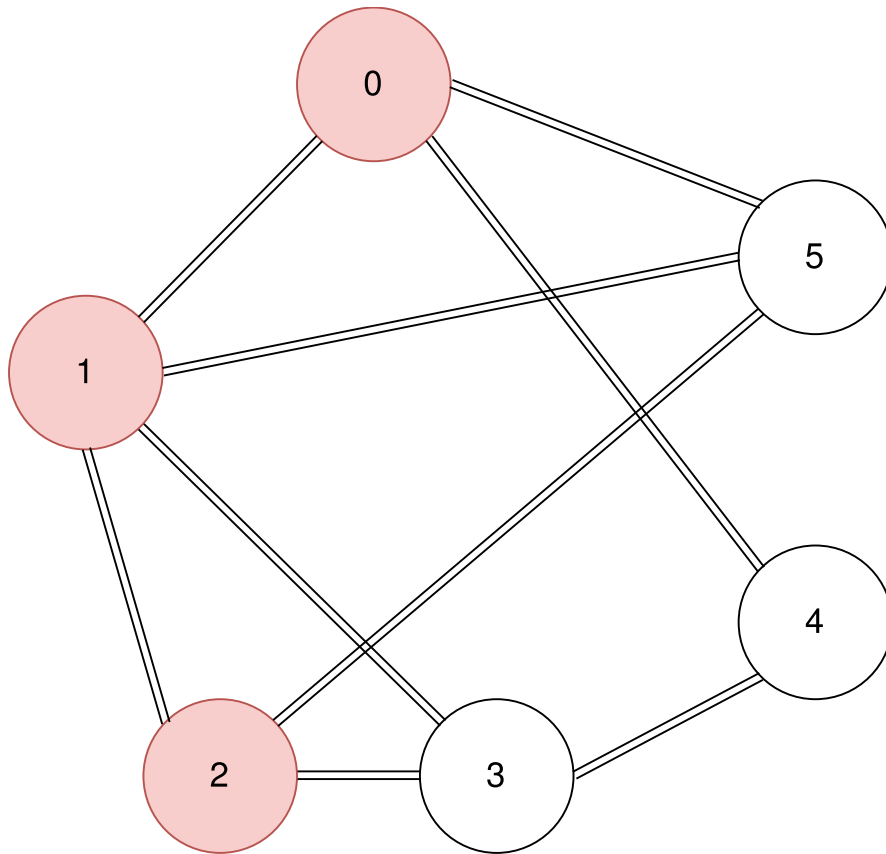
(1) 访问节点 0 本身，将它染成红色，在其邻节点 {1, 5} 中挑选节点 1，继续遍历；



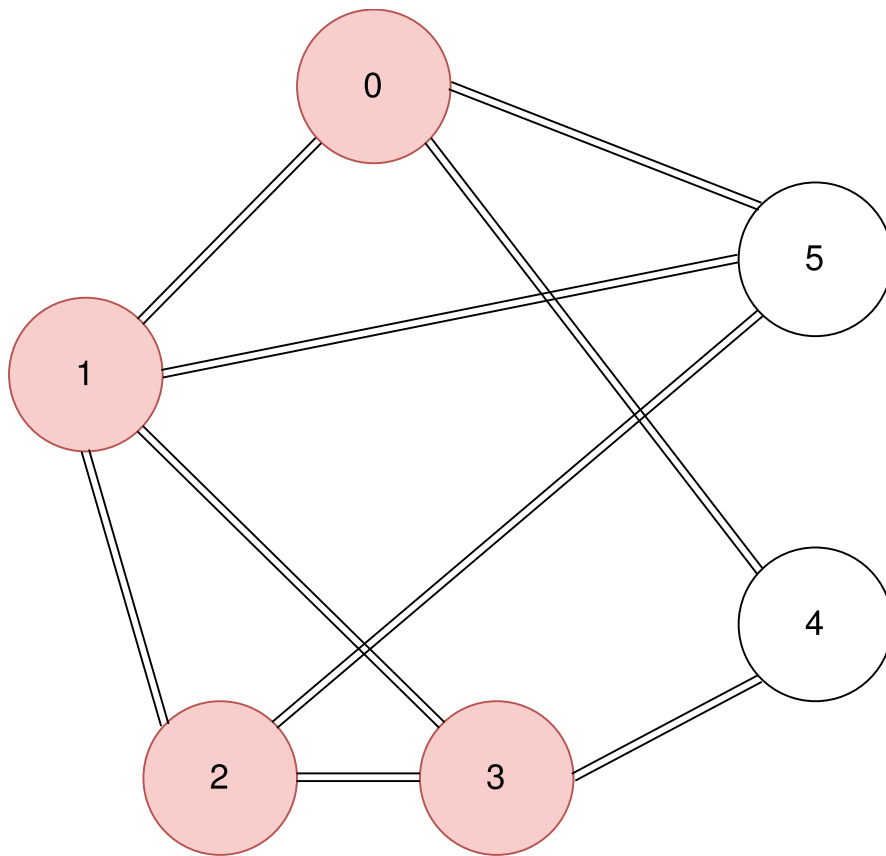
(2) 访问节点 1 本身，将它染成红色，其邻节点 $\{0, 2\}$ 中由于节点 0 已经为红色，因此不再考虑该节点，挑选节点 2，继续遍历；



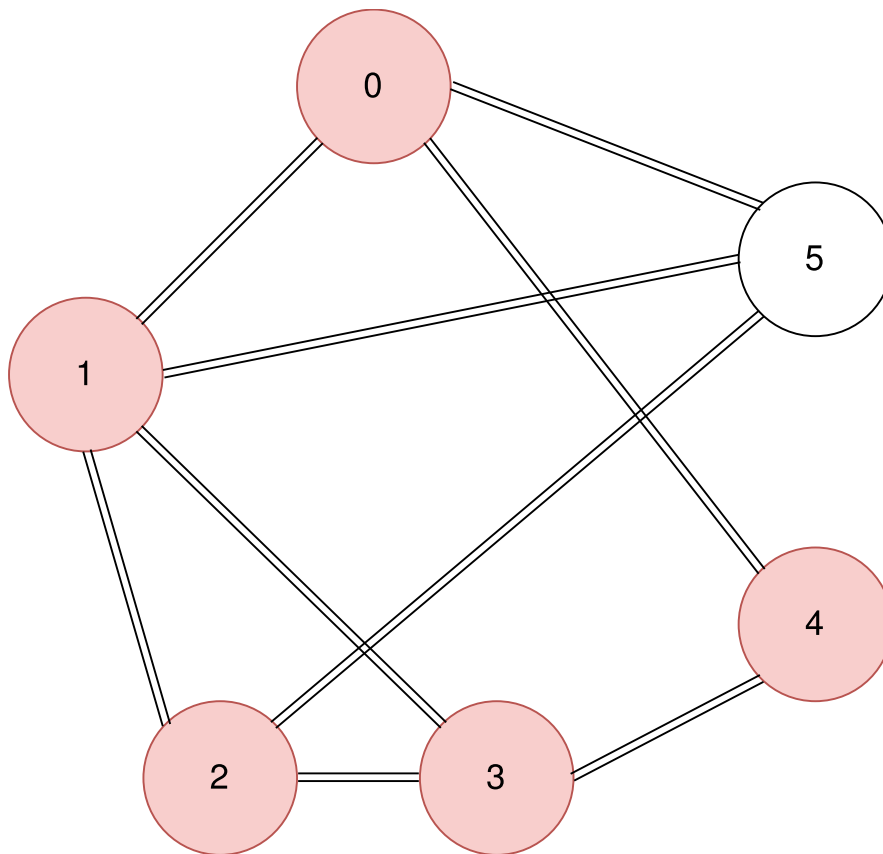
(3) 访问节点 2 本身，将它染成红色，其邻节点 $\{1, 3\}$ 中由于节点 1 已经为红色，因此不再考虑该节点，挑选节点 3，继续遍历；



(4) 访问节点 3 本身，将它染成红色，其邻节点 {2, 4} 中由于节点 2 已经为红色，因此不再考虑该节点，挑选节点 4，继续遍历；



(5) 访问节点 4 本身，将它染成红色，其邻节点 $\{0, 3\}$ 中的所有节点都已经为红色，遍历结束，返回上一层；



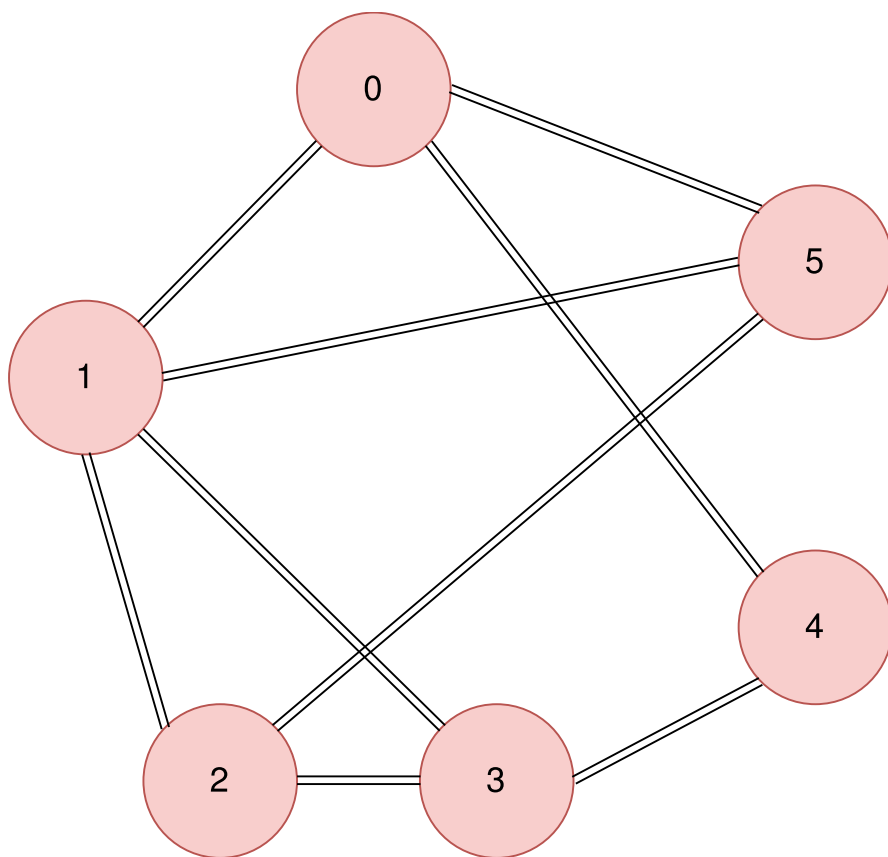
(6) 上一层节点 3 的遍历中，其邻节点 {2, 4} 中的所有节点都已经为红色，遍历结束，返回上一层；

(7) 上一层节点 2 的遍历中，其邻节点 {1, 3} 中的所有节点都已经为红色，遍历结束，返回上一层；

(8) 上一层节点 1 的遍历中，其邻节点 {0, 2} 中的所有节点都已经为红色，遍历结束，返回上一层；

(9) 上一层节点 0 的遍历中，其邻节点 {1, 5} 中节点 1 已经为红色，不再考虑，挑选节点 5，继续遍历；

(10) 访问节点 5 本身，将它染成红色，其邻节点 {0, 2} 中的所有节点都已经为红色，遍历结束，返回上一层；



(11) 上一层节点 0 的遍历中，其邻节点 {1, 5} 中的所有节点都已经为红色，遍历结束，算法结束；

深度优先搜索的时间复杂度是 $O(n)$ 。

深度优先搜索

- https://en.wikipedia.org/wiki/Depth-first_search

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



BreadthFirstSearch 广度优先搜索

- [Breadth First Search\(BFS\) - 广度优先搜索](#)
 - [问题](#)
 - [解法](#)
 - [广度优先搜索](#)
 - [源码](#)
 - [测试](#)

Breadth First Search(BFS) - 广度优先搜索

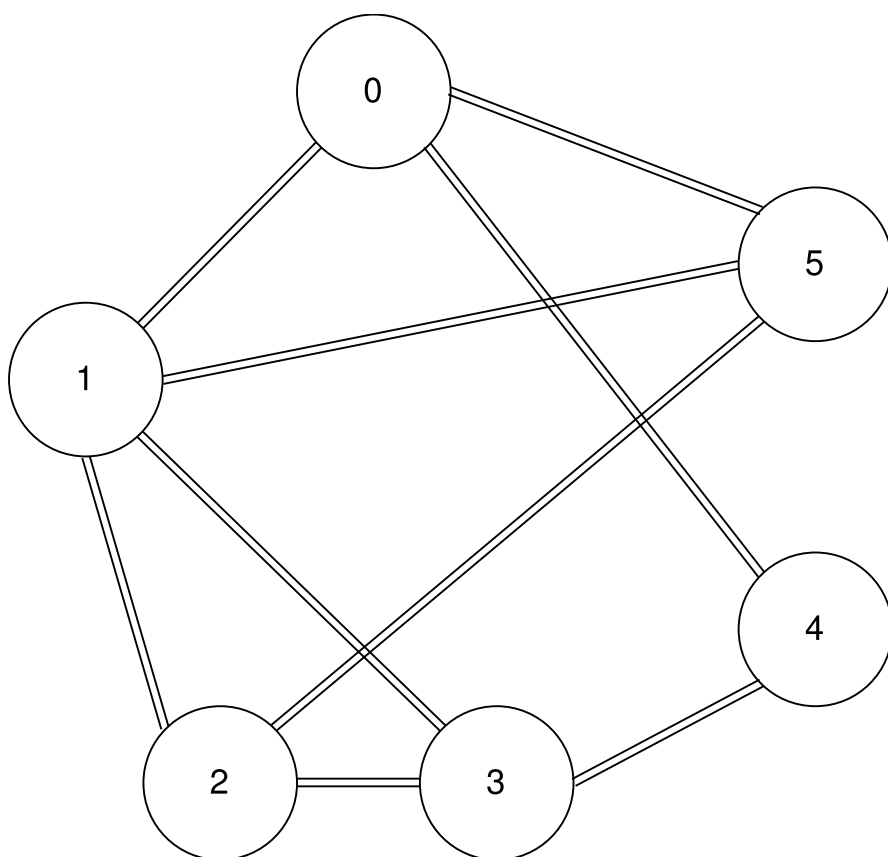
问题

用广度优先搜索从图 G 的节点 beg 开始，遍历图 G 中的所有节点。

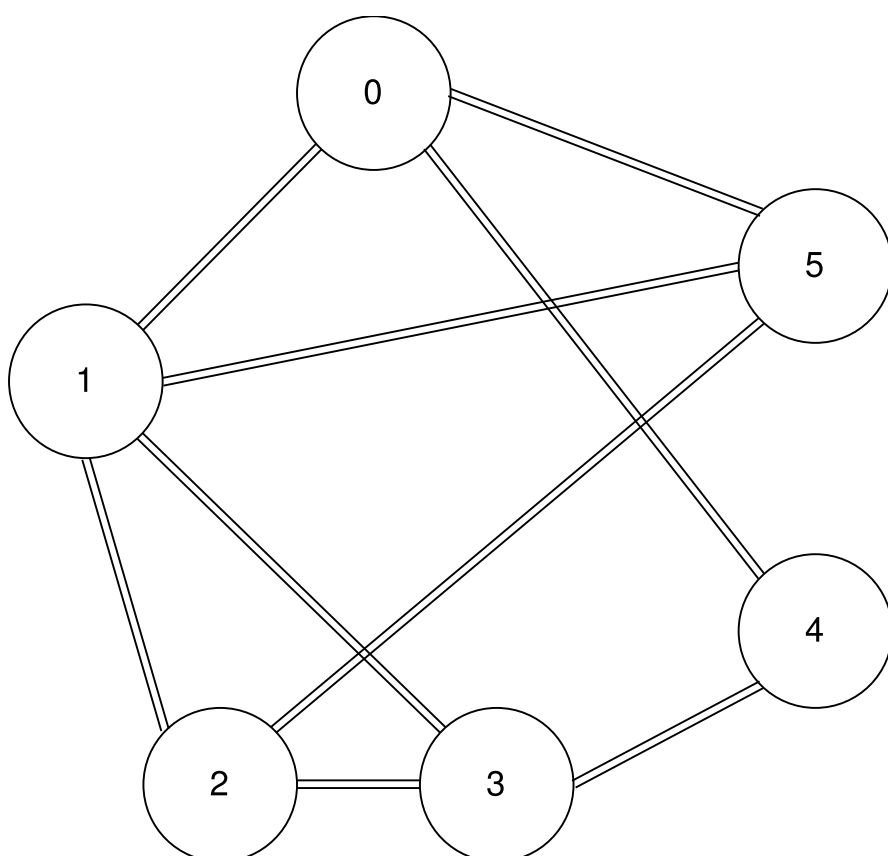
解法

在图 G 中，假设节点 i 的邻节点集合为 V_i ，对于图中的任意节点 i ，在访问节点 i 之后，总是优先访问该节点的邻节点集合 V_i 中的所有节点，然后才继续访问其他节点。

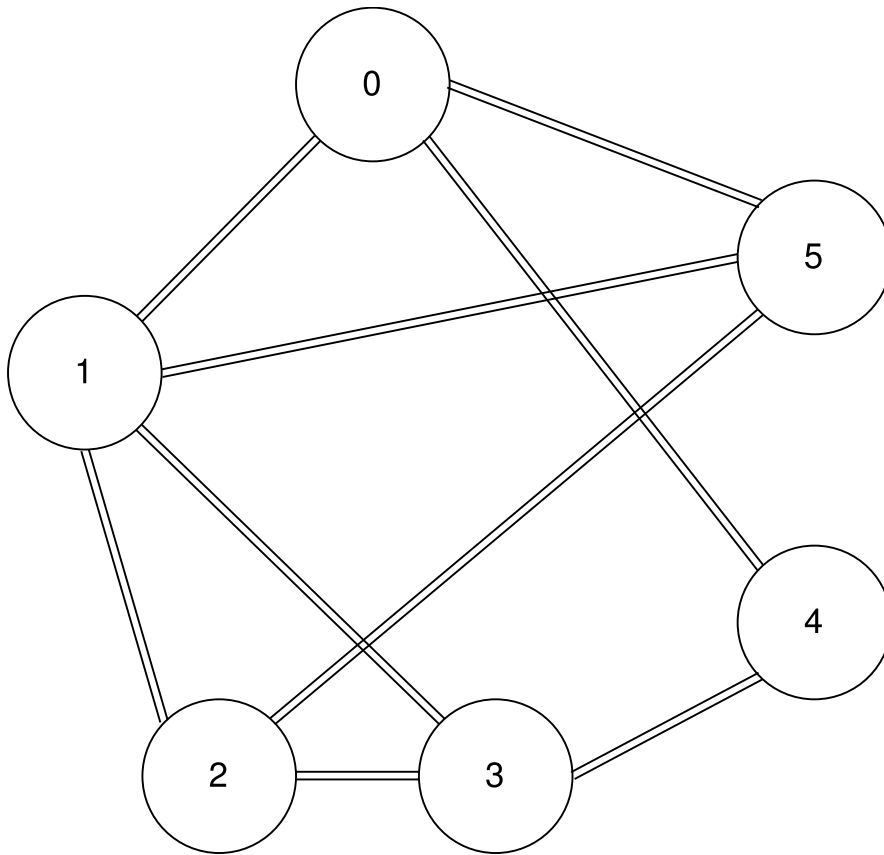
广度优先遍历需要一个队列 `queue` 来存储那些等待访问而尚未被访问的节点，在遍历过程中，为了避免重复的访问一个节点，当某个节点 i 加入 `queue` 时我们将其染成红色。下面演示从无向图 G 中的节点 0 开始进行广度优先搜索过程：



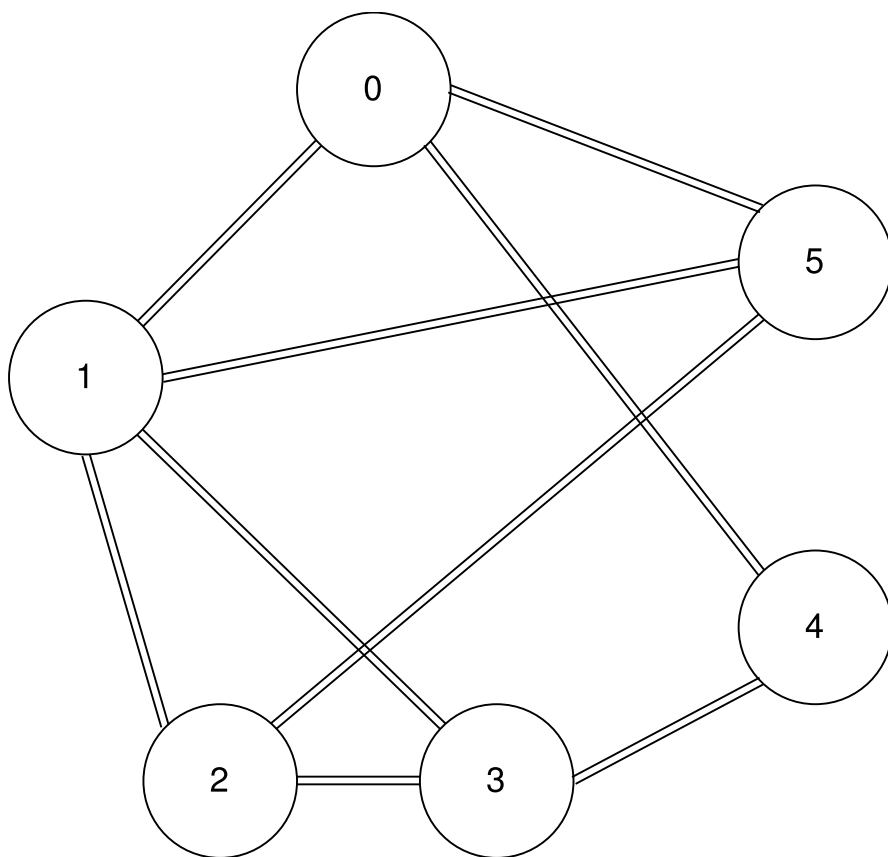
(1) 初始时从节点 0 开始，将它染红并加入 queue 中；



(2) 从 queue 中取出头节点 0 进行访问，然后考虑其未被染色的邻节点 {1, 4, 5}，将 {1, 4, 5} 节点染红并加入 queue 中；



(3) 从 queue 中取出头节点 1 进行访问，然后考虑其未被染色的邻节点 {2, 3}，将 {2, 3} 节点染红并加入 queue 中；



(4) 从 queue 中取出头节点 4 进行访问，它没有未被染色的邻节点；

(5) 从 queue 中取出头节点 5 进行访问，它没有未被染色的邻节点；

(6) 从 queue 中取出头节点 2 进行访问，它没有未被染色的邻节点；

(7) 从 queue 中取出头节点 3 进行访问，它没有未被染色的邻节点；

(8) 队列 queue 为空，算法结束；

广度优先搜索的时间复杂度是 $O(n)$ 。

广度优先搜索

- https://en.wikipedia.org/wiki/Breadth-first_search

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

TopologicalSort 拓扑排序

- [Topological Sort - 拓扑排序](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Topological Sort - 拓扑排序

问题

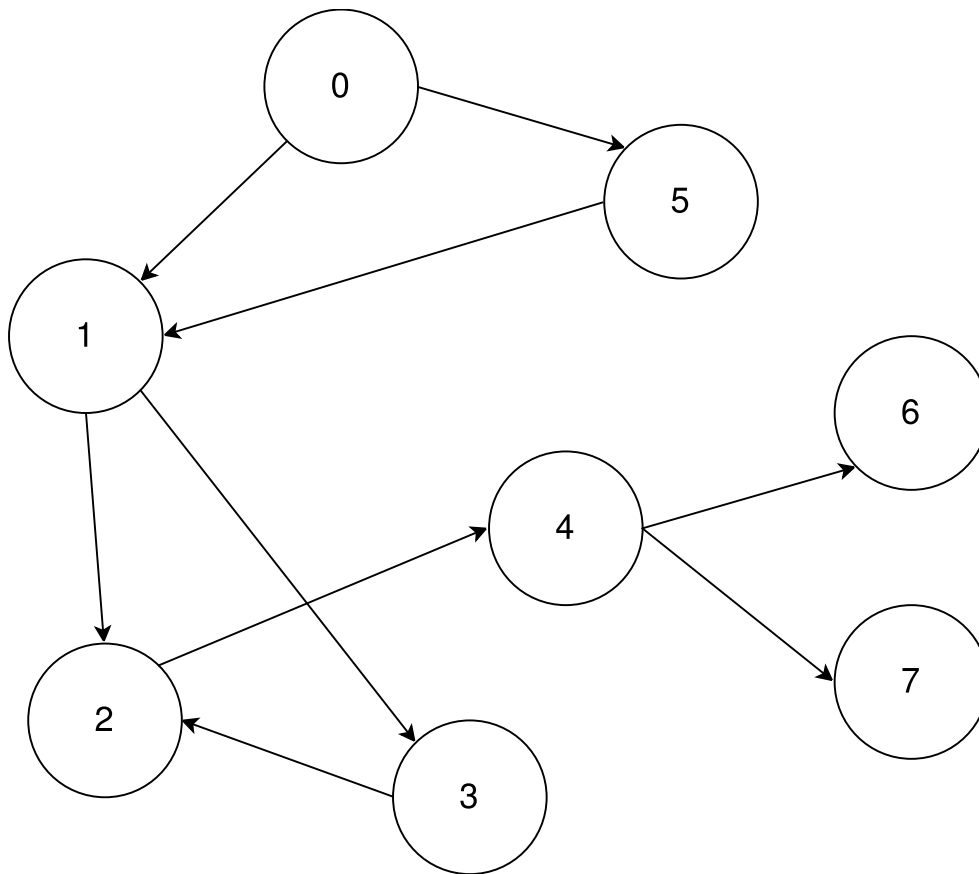
对有向图 G 进行拓扑排序。

解法

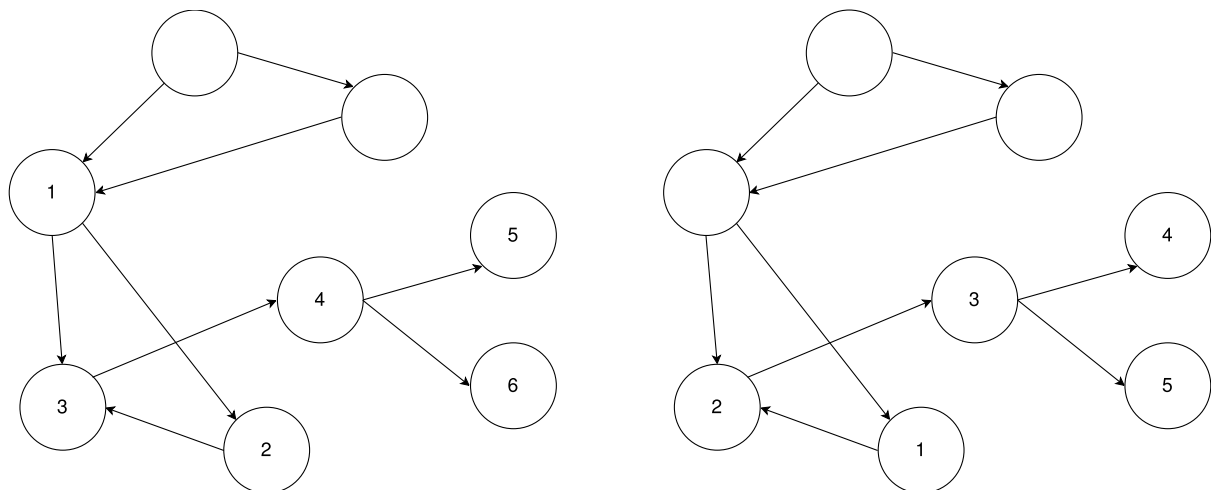
拓扑排序可以通过应用深度优先搜索来解决。

对于有向图 G 中的每个节点 i ，都进行一次深度优先搜索，由于DFS的特性，每递归一次都尝试让节点 i 走的更远，直到终点。因此从节点 i 出发DFS所经过的节点数量可看作是节点 i 到终点的距离 d 。然后按照距离 d 对所有节点进行排序即可得到拓扑排序。其中将终点到自己的距离作为 1 。

下面以有向图 G 作为一个例子进行拓扑排序：



从节点 4 开始可以遍历到 6 个节点，从节点 3 开始可以遍历到 5 个节点：



对图中的所有节点依次进行DFS，节点 0 - 7 到终点的距离依次为 8, 6, 3, 4, 2, 7, 1, 1。因此按照距离排序后，有向图G的拓扑排序为[0, 5, 1, 3, 2, 4, 6, 7]。

拓扑先序遍历的时间复杂度是 $O(n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```


EulerCycle 欧拉回路

- [Euler Cycle - 欧拉回路](#)
 - [问题](#)
 - [无向图 \$UG\$ 的欧拉回路解法](#)
 - [有向图 \$DG\$ 的欧拉回路解法](#)
 - [源码](#)
 - [测试](#)

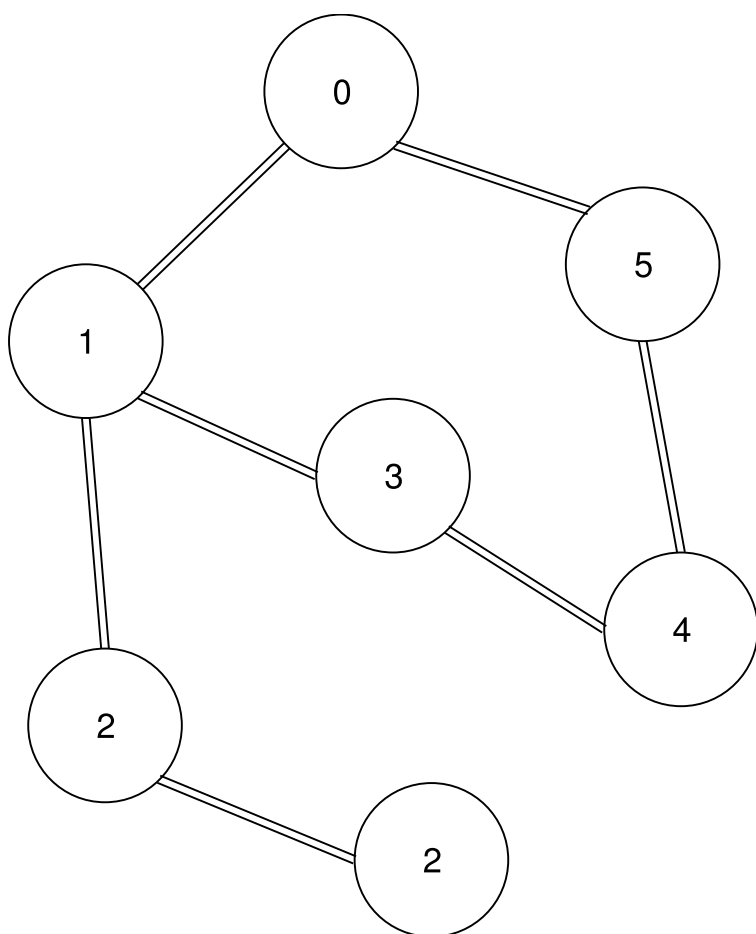
Euler Cycle - 欧拉回路

问题

求无向图 UG 和有向图 DG 的欧拉回路。

无向图 UG 的欧拉回路解法

本文介绍求无向图欧拉回路的 Fleury 算法。我们假定本问题给定的无向图 UG 中必然存在欧拉回路（因为欧拉回路存在的判定非常简单）。设矩阵 g 表示无向图 UG ，其中 $g[i, j] = 1$ 表示顶点 v_i 到 v_j 之间存在单向边 $e \{i, j\}$ ， $g[i, j] = 0$ 表示顶点 v_i 到 v_j 之间不存在单向边 $e \{i, j\}$ 。



上面的无向图可以表示为

```

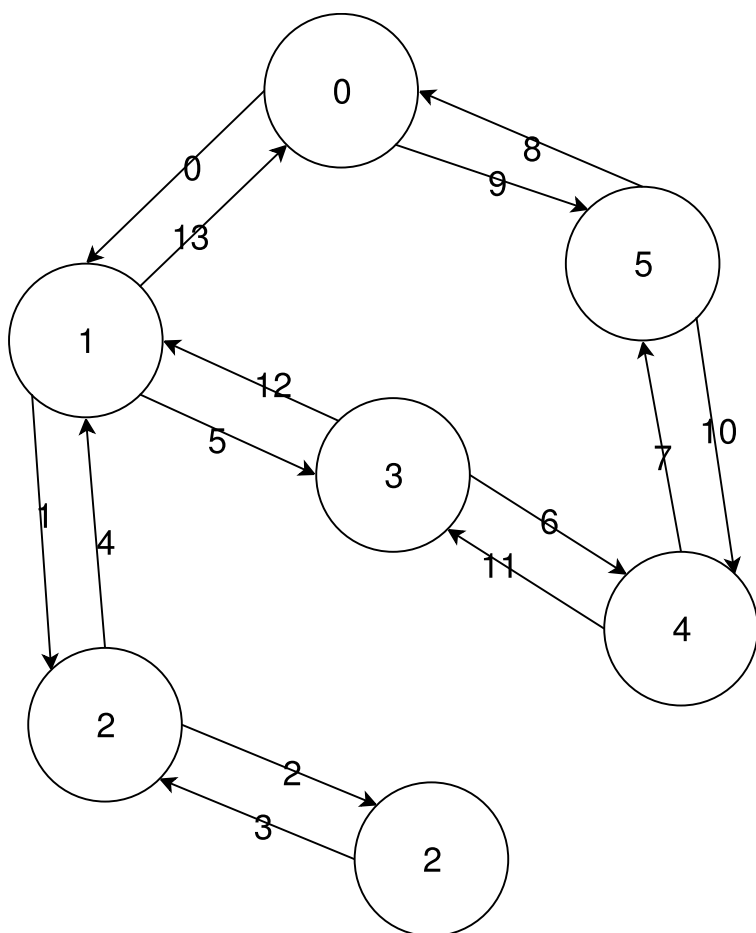
g =
\begin{bmatrix}
0\{0,0\} & 1\{0,1\} & 0\{0,2\} & 0\{0,3\} & 0\{0,4\} & 1\{0,5\} & 0\{0,6\} \setminus \\
1\{1,0\} & 0\{1,1\} & 1\{1,2\} & 1\{1,3\} & 0\{1,4\} & 0\{1,5\} & 0\{1,6\} \setminus \\
0\{2,0\} & 1\{2,1\} & 0\{2,2\} & 0\{2,3\} & 0\{2,4\} & 0\{2,5\} & 0\{2,6\} \setminus \\
0\{3,0\} & 1\{3,1\} & 0\{3,2\} & 0\{3,3\} & 1\{3,4\} & 0\{3,5\} & 0\{3,6\} \setminus \\
0\{4,0\} & 0\{4,1\} & 0\{4,2\} & 1\{4,3\} & 0\{4,4\} & 1\{4,5\} & 0\{4,6\} \setminus \\
1\{5,0\} & 0\{5,1\} & 0\{5,2\} & 0\{5,3\} & 1\{5,4\} & 0\{5,5\} & 0\{5,6\} \setminus \\
0\{6,0\} & 0\{6,1\} & 1\{6,2\} & 0\{6,3\} & 0\{6,4\} & 0\{6,5\} & 0\{6,6\} \setminus
\end{bmatrix}

```

随机的选取任意顶点（这里我们选择节点 v_0 ）作为起始顶点，对整个图进行一种类似 *DFS* 搜索的操作。对于顶点 v_i ，在它的所有邻节点中选择任意节点 v_j 作为下一个遍历的节点。但需要保证：必须存在一条与边 $e_{\{i,j\}}$ 相反的边 $e_{\{j,i\}}$ ，来保证可以从 v_i 到达 v_j ，又可以从 v_j 到达 v_i 。即必须保证 $g[i,j] = 1$ 且 $g[j,i] = 1$ 。

在进行上面的遍历操作过程中，从节点 v_i 移动到邻节点 v_j 时，将边 $e_{\{i,j\}}$ 删去（令

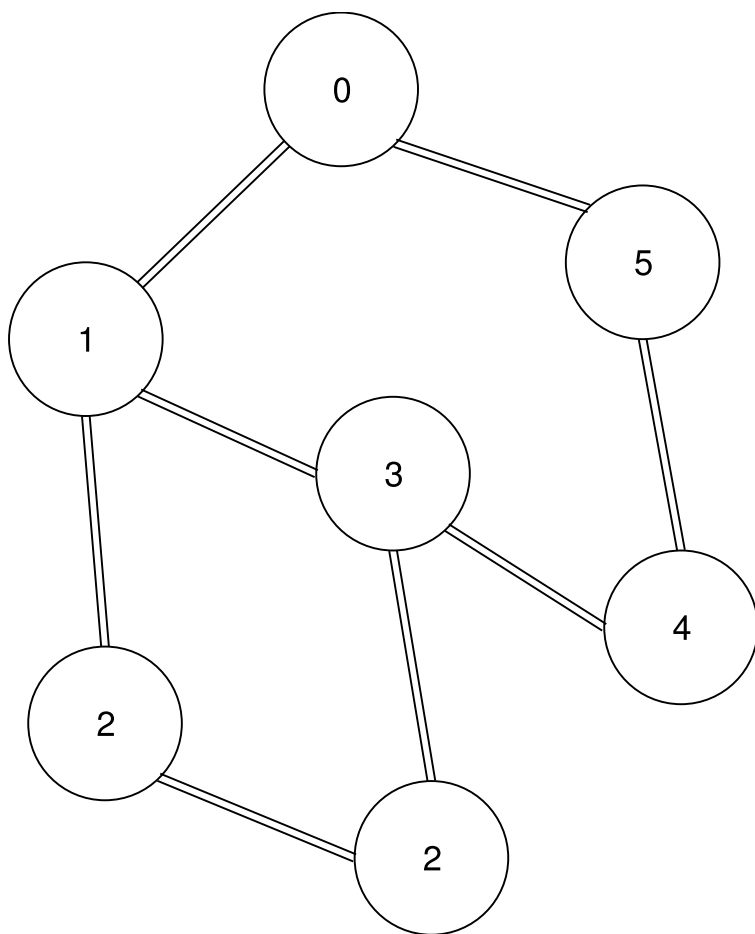
$g[i, j] = 0$)。重复这样的遍历操作，直到再次回到起始节点 v_0 ，算法结束。整个过程中删除的边的顺序即为一条欧拉回路，可以用一个链表依次记录所有被删除的边。上面无向图的遍历过程如下，每条边上的数字代表该条边被经过的顺序。



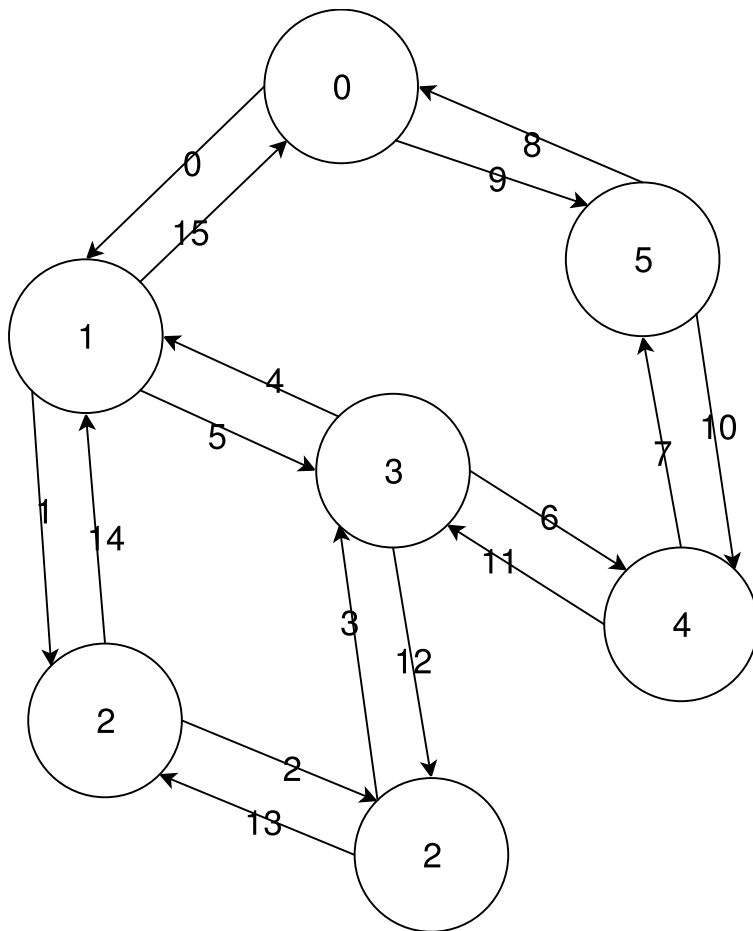
在上面删除边的操作中，当从 v_1 移动到 v_2 时，令 $g[1, 2] = 0$ （即删除边 e_1 ），之后还会出现从 v_2 移动到 v_1 的情况，这时边 e_4 不存在与它相反的边 e_1 （之前被删除了）。这时节点 v_2 相邻的节点中没有满足上述条件这样的两条相反的边，这时可以打破该原则，仍然选择边 e_4 ，从 v_2 移动到 v_1 。

优先选择两条相反的边的临节点，该原则是为了保证存在一条可以返回的路径。当图中不存在这些双向边的时候，说明图中已经不存在双向的边，这个原则就可以打破，不再考虑了。最后，在所有只有单向的边中，尽量最后选择返回起始节点的边，因为那条边应该是欧拉回路中的最后一条边。

下图是在上图的基础上增加了边 $e_{\{3, 6\}}$ ：



在遍历该图的时候，当从节点 v_3 移动到 v_1 时，这时可以选择的边有 e_5 和 e_{15} 。这时应该最后选择返回起点 v_0 的边，所以应该放弃边 e_{15} ，选择 e_5 。最终遍历的顺序如下：



Fleury 算法的时间复杂度是 $O(|E|)$ ($|E|$ 为无向图 UG 的边的数量)。

有向图 DG 的欧拉回路解法

有向图也可以用 Fleury 算法来进行求解，唯一的不同就是不能随意选择任意节点作为起始点，因为有向图中的欧拉回路与无向图不一样。有向图的欧拉回路起点满足 $\text{degree}\{out\} = \text{degree}\{in\} + 1$ ，终点满足 $\text{degree}\{in\} = \text{degree}\{out\} + 1$ 。因此需要遍历有向图 DG 找出欧拉回路的起点，再应用 Fleury 算法。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```


Section-2 MinimumSpanningTree 第2节 最小生成树

KnowledgePoint 知识要点

- [Knowledge Point - 知识要点](#)
 - [最小生成树 - Minimum Spanning Tree](#)

Knowledge Point - 知识要点

最小生成树 - Minimum Spanning Tree

无向图 G 中的每个边都有一个权值，存在这样一条路径（边的集合），它没有回路，连接图 G 中所有顶点，且集合中边的权值之和最小，则该路径称为图 G 的最小生成树（这里的最小指经过的边的权值之和）。

Kruskal Kruskal算法

- [Kruskal - Kruskal算法](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Kruskal - Kruskal算法

问题

用Kruskal算法求无向图 G 的最小生成树。

解法

Kruskal算法是一种贪心算法。初始时将图 G 的边集 E 按照权值，从小到大进行排序，并且生成树。从最小权值的边开始，依次考虑每一条边，对于边 e_i 来说，若将它加入生成树集合 S 中， e_i 不会与 S 中已有的边形成环，那么选取边 e_i 作为生成树中的一条边，将其加入集合 S ；反之若将 e_i 加入 S 中会与已有的边形成环，则跳过 e_i 考虑下一个。

这里对于环的判断，比较像<DisjointSet 并查集>的算法思路：

判断一条边 e_i 加入生成树集合 S 是否会出现回路，不需要在生成树集合 S 中判断回路是否存在（太过麻烦）。判断边 e_i 的两个端点 v_1 和 v_2 是否都属于集合 S 。若两个端点都属于集合 S ，则添加边 e_i 会使生成树中增加环，因此需要跳过该边。下图中红色的边已经加入生成树集合 S 中，节点 v_2 和 v_3 属于生成树集合 S ，因此边 $e_{\{2-3\}}$ 加入生成树中会产生环：

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```


Prim Prim算法

- [Breadth First Search\(BFS\)](#)
- [广度优先搜索](#)
- [Upper Folder - 上一级目录](#)
- [Source Code - 源码](#)
- [Test Code - 测试](#)

Breadth First Search(BFS)

广度优先搜索

问题：

用广度优先搜索从图(G)的节点(beg)开始，遍历图(G)中的所有节点。

解法：

在图(G)中，假设节点(i)的邻节点集合为(V_i)，对于图中的任意节点(i)，在访问节点(i)之后，总是优先访问该节点的邻节点集合(V_i)中的所有节点，然后才继续访问其他节点。

广度优先遍历需要一个队列(queue)来存储那些等待访问而尚未被访问的节点，在遍历过程中，为了避免重复的访问一个节点，当某个节点(i)加入(queue)时我们将其染成红色。下面演示从无向图(G)中的节点(0)开始进行广度优先搜索过程：



((1))初始时从节点(0)开始，将它染红并加入(queue)中；



((2))从(queue)中取出头节点(0)进行访问，然后考虑其未被染色的邻节点({1, 4, 5})，将({ 1, 4, 5 })节点染红并加入(queue)中；



((3))从(queue)中取出头节点(1)进行访问, 然后考虑其未被染色的邻节点({2, 3}), 将({2, 3})节点染红并加入(queue)中;



((4))从(queue)中取出头节点(4)进行访问, 它没有未被染色的邻节点;

((5))从(queue)中取出头节点(5)进行访问, 它没有未被染色的邻节点;

((6))从(queue)中取出头节点(2)进行访问, 它没有未被染色的邻节点;

((7))从(queue)中取出头节点(3)进行访问, 它没有未被染色的邻节点;

((8))队列(queue)为空, 算法结束;

广度优先搜索的时间复杂度是($O(n)$)。

广度优先搜索:

- https://en.wikipedia.org/wiki/Breadth-first_search

SecondMinimumSpanningTree 次小生成树

- [Breadth First Search\(BFS\)](#)
- [广度优先搜索](#)
- [Upper Folder](#) - 上一级目录
- [Source Code](#) - 源码
- [Test Code](#) - 测试

Breadth First Search(BFS)

广度优先搜索

问题：

用广度优先搜索从图(G)的节点(beg)开始，遍历图(G)中的所有节点。

解法：

在图(G)中，假设节点(i)的邻节点集合为(V_i)，对于图中的任意节点(i)，在访问节点(i)之后，总是优先访问该节点的邻节点集合(V_i)中的所有节点，然后才继续访问其他节点。

广度优先遍历需要一个队列(queue)来存储那些等待访问而尚未被访问的节点，在遍历过程中，为了避免重复的访问一个节点，当某个节点(i)加入(queue)时我们将其染成红色。下面演示从无向图(G)中的节点(0)开始进行广度优先搜索过程：



((1))初始时从节点(0)开始，将它染红并加入(queue)中；



((2))从(queue)中取出头节点(0)进行访问，然后考虑其未被染色的邻节点({1, 4, 5})，将({ 1, 4, 5 })节点染红并加入(queue)中；



((3))从(queue)中取出头节点(1)进行访问, 然后考虑其未被染色的邻节点({2, 3}), 将({2, 3})节点染红并加入(queue)中;



((4))从(queue)中取出头节点(4)进行访问, 它没有未被染色的邻节点;

((5))从(queue)中取出头节点(5)进行访问, 它没有未被染色的邻节点;

((6))从(queue)中取出头节点(2)进行访问, 它没有未被染色的邻节点;

((7))从(queue)中取出头节点(3)进行访问, 它没有未被染色的邻节点;

((8))队列(queue)为空, 算法结束;

广度优先搜索的时间复杂度是($O(n)$)。

广度优先搜索:

- https://en.wikipedia.org/wiki/Breadth-first_search

OptimalRatioSpanningTree 最优比率生成树

- [Breadth First Search\(BFS\)](#) - 广度优先搜索
 - [问题](#)
 - [解法](#)

Breadth First Search(BFS) - 广度优先搜索

问题

用广度优先搜索从图(G)的节点(beg)开始, 遍历图(G)中的所有节点。

解法

在图(G)中, 假设节点(i)的邻节点集合为(V_i), 对于图中的任意节点(i), 在访问节点(i)之后, 总是优先访问该节点的邻节点集合(V_i)中的所有节点, 然后才继续访问其他节点。

广度优先遍历需要一个队列(queue)来存储那些等待访问而尚未被访问的节点, 在遍历过程中, 为了避免重复的访问一个节点, 当某个节点(i)加入(queue)时我们将其染成红色。下面演示从无向图(G)中的节点(0)开始进行广度优先搜索过程:



((1))初始时从节点(0)开始, 将它染红并加入(queue)中;



((2))从(queue)中取出头节点(0)进行访问, 然后考虑其未被染色的邻节点({1, 4, 5}), 将({ 1, 4, 5 })节点染红并加入(queue)中;



((3))从(queue)中取出头节点(1)进行访问, 然后考虑其未被染色的邻节点({2, 3}), 将({ 2, 3 })节点染红并加入(queue)中;



((4))从(queue)中取出头节点(4)进行访问, 它没有未被染色的邻节点;

((5))从(queue)中取出头节点(5)进行访问, 它没有未被染色的邻节点;

((6))从(queue)中取出头节点(2)进行访问, 它没有未被染色的邻节点;

((7))从(queue)中取出头节点(3)进行访问, 它没有未被染色的邻节点;

((8))队列(queue)为空, 算法结束;

广度优先搜索的时间复杂度是($O(n)$)。

广度优先搜索:

- https://en.wikipedia.org/wiki/Breadth-first_search
- [Upper Folder](#) - 上一级目录
- [Source Code](#) - 源码
- [Test Code](#) - 测试

Section-3 ShortestPath 第3节 最短路径

KnowledgePoint 知识要点

- [KnowledgePoint 知识要点](#)

KnowledgePoint 知识要点

Relaxation 松弛操作

- [Relaxation 松弛操作](#)

Relaxation 松弛操作

BellmanFord BellmanFord算法

- [BellmanFord BellmanFord算法](#)

BellmanFord BellmanFord算法

ShortestPathFasterAlgorithm 最短路径更快算法 (SPFA)

- [ShortestPathFasterAlgorithm 最短路径更快算法 \(SPFA\)](#)

ShortestPathFasterAlgorithm 最短路径更快算法 (SPFA)

Dijkstra Dijkstra算法

- [Dijkstra Dijkstra算法](#)

Dijkstra Dijkstra算法

Floyd Floyd算法

- [Floyd Floyd算法](#)

Floyd Floyd算法

DifferentConstraints 差分约束

- [DifferentConstraints 差分约束](#)

DifferentConstraints 差分约束

Section-4 Connectivity 第4节 连通

KnowledgePoint 知识要点

- [知识要点](#)

知识要点

平凡图 - Trivial Graph:

只有一个节点，没有边的图。

非平凡图 - Nontrivial Graph:

有至少两个节点，一条边的图。

连通 - Connectivity:

若图(G)中从顶点(v_i)到另一顶点(v_j)有路径相连，并且从(v_j)也可以移动到(v_i)，则称(v_i)和(v_j)是连通的（在无向图中，若(v_i)可以到达(v_j)，则显然(v_j)也可以到达(v_i)）。

连通图 - Connected Graph:

若图(G)中从任意两个顶点(v_i)和(v_j)都是连通的，则称图(G)是连通图。

极大连通子图 - Strong Components:

若图(G)中的子图(S)是一个连通图，且不论加入任意其他节点，都会使得子图(S)不再连通，则称(S)为极大连通子图。连通图的极大连通子图即为它自己，非连通图中存在多个极大连通子图。

连通分量（连通分支） - Connected Component，强连通分量（强连通分支） - Strongly Connected Component:

连通分量是无向图(G)中的一个极大连通子图，其中的任意节点(v_i)可以到达任意节点(v_j)，且(v_j)也可以到达(v_i)。类似的，强连通分量（强连通分支）是有向图(G)中的一个极大连通子图，其中的任意节点(v_i)可以到达任意节点(v_j)，且(v_j)也可以到达(v_i)。求强连通分量的算法有双向遍历后取交集、*Tarjan*算法、*Kosaraju*算法、*Gabow*算法

逆图：

图($G = (V, E)$)的逆图，是将其所有边都进行反向，得到($G' = (V', E')$)。点集(V')与(V)相同，而边集(E')的任意边($e' = (v, u)$)都在(E)中存在对应的边($e = (u, v)$)。逆图(G')也称为图(G)的转置。无向图的逆图永远都是它自己。

割 - *Cut*：

割($C = (S, T)$)是图($G = (V, E)$)中点集 V 的划分，边($e = (u, v)$)中(u)属于子图(S)，(v)属于子图(T)，且 S 和 T 互不相交。割集(*cut-set*)是割的集合($\{ (u, v) \in E \mid u \in S, v \in T \}$)，其中(S)和(T)是互不相交的子图。

点割集：

图($G = (V, E)$)的点集(V)中存在一个非空真子集(V_1)，满足(G)中删去(V_1)后不再连通。但对于(V_1)的任意真子集(V_2)，(G)中删去(V_2)后都仍然连通，称这样的点集(V_1)是(G)的一个点割集。非平凡的无向连通图存在点割集的充要条件是该图中至少存在两个不相邻的相异节点。

边割集：

图($G = (V, E)$)的边集(E)中有这样的非空真子集(E_1)，(G)中删去(E_1)后不再连通。但对于(E_1)的任意真子集(E_2)，(G)中删去(E_2)后都仍然连通，称这样的边集(E_1)是(G)的一个边割集。任何非平凡的无向连通图一定具有边割集。

点连通度：

非平凡的无向连通图(G)的所有点割集($V\{cut\}$)中的节点数量最少的那个点割集(V_1)，其节点数量(k)即为图(G)的点连通度。即图(G)删除任意($k-1$)个节点后都仍然能够连通，但存在一个方案（点

割集)使删除(k)个节点后不连通。特别的,当($k = 2$)时,即图(G)任意删除(1)个节点都仍然能够连通,但存在一个方案,删除(2)个节点后不再连通,称点连通度($k = 2$)的连通分支(图)为点双连通分支。

边连通度:

非平凡的无向连通图(G)的所有边割集($E\{cut\}$)中的边数量最少的那个边割集(E_1),其边数量(k)即为图(G)的边连通度,即图(G)删去任意($k-1$)条边后都仍然能够连通,但存在一个方案(边割集)使删除 k 条边后不再连通。特别的,当 $k = 2$ 时,即该图任意删除1条边都仍然连通,但存在一个方案使删除2条边后不再连通,称边连通度 $k = 2$ 的连通分支(图)为边双连通分支。

割点:

无向图(G)中存在某个点(v_1),删去该点后图的连通分支数量加1,即该点将原图分成两个连通分支,则称点(v_1)为无向图(G)的割点。假设删去一个割点后得到两个新的连通分支,则割点可以看作同时属于两个新连通分支,而非割点总是只属于一个连通分支。

割边(桥):

无向图(G)中存在某条边(e_1),删去该边后图的连通分支数量加1,即该边将原图分成两个连通分支。假设删去一条割边后得到两个新的连通分支,则割边的两端点可以看作分别属于两个新连通分支,而非割边的两端点只属于同一个连通分支。

双连通分支 - Double Connected Component:

有向图(G)的一个极大连通子图,强连通分量中的任意节点(v_i)可以到达任意节点(v_j),且(v_j)也可以到达(v_i)。求强连通分量的算法有双向遍历后取交集、Tarjan算法、Kosaraju算法、Gabow算法

Kosaraju Kosaraju算法

- [Kosaraju算法](#)

Kosaraju算法

问题：

用Kosaraju算法求有向图(G)的强连通分支。

解法：

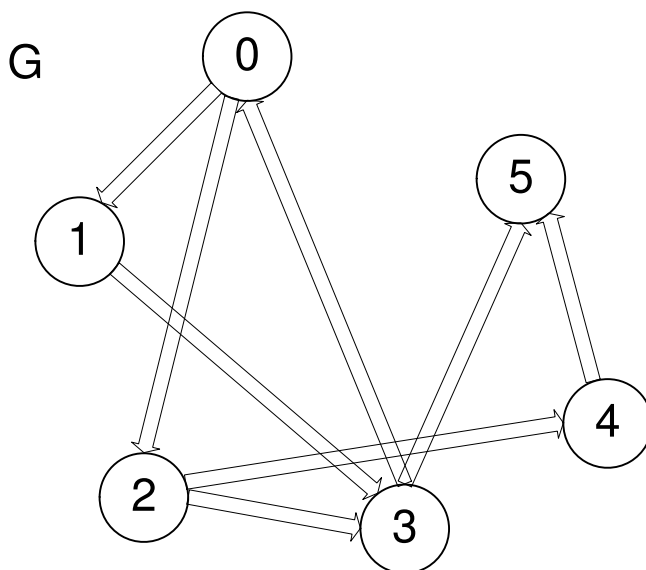
Kosaraju算法分为3个步骤：

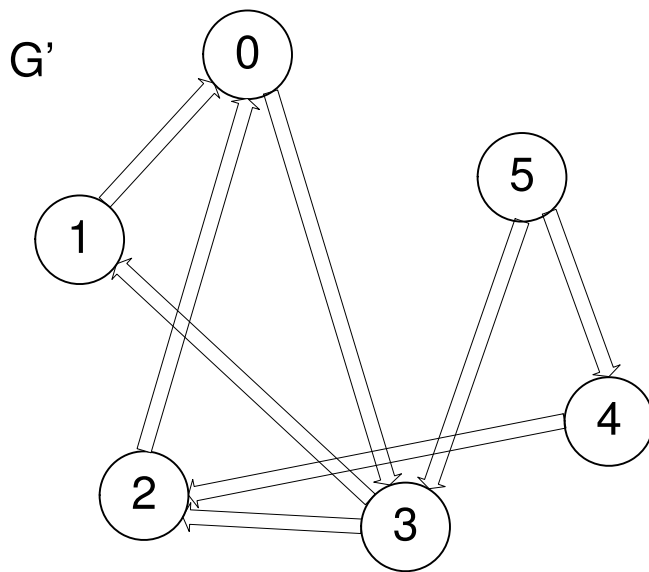
((1)) 求出图(G)的逆图(G')；

((2)) 求出逆图(G')的拓扑排序(T)；

((3)) 按照逆图(G')拓扑排序(T)的顺序，对原图(G)中每个节点进行DFS，得到的森林即为所求的强连通分支。注意每个节点只能属于一个强连通分支，因此当一个节点被DFS遍历到后，需要将其染红；

求出有向图(G)的逆图(G')：





用矩阵可以表示为：

```
[
G =
\begin{bmatrix}
0 & 1 & 1 & 0 & 0 & 0 \backslash \\
0 & 0 & 0 & 1 & 0 & 0 \backslash \\
0 & 0 & 0 & 1 & 1 & 0 \backslash \\
1 & 0 & 0 & 0 & 0 & 1 \backslash \\
0 & 0 & 0 & 0 & 0 & 1 \backslash \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
G' =
\begin{bmatrix}
0 & 0 & 0 & 1 & 0 & 0 \backslash \\
1 & 0 & 0 & 0 & 0 & 0 \backslash \\
1 & 0 & 0 & 0 & 0 & 0 \backslash \\
0 & 1 & 1 & 0 & 0 & 0 \backslash \\
0 & 0 & 1 & 0 & 0 & 0 \backslash \\
0 & 0 & 0 & 1 & 1 & 0 \\
\end{bmatrix}
]
```

对逆图(G')进行拓扑排序，得到顺序($T = [5, 4, 0, 1, 2, 3]$)。

按照顺序(T)对原图(G)进行DFS搜索,得到下面的森林,即为有向图(G)的(3)个强连通分支。这(3)个强连通分支中,任意节点(v_i)都存在一条路径可以到达其他任意节点(v_j)。

```
[  
  \begin{bmatrix}  
    tree_1 = [ 5 ] \  
    tree_2 = [ 4 ] \  
    tree_3 = [ 0, 1, 3, 2 ]  
  \end{bmatrix}  
]
```

Tarjan Tarjan算法

Tarjan算法

问题：

用Tarjan算法求有向图(G)的强连通分支。

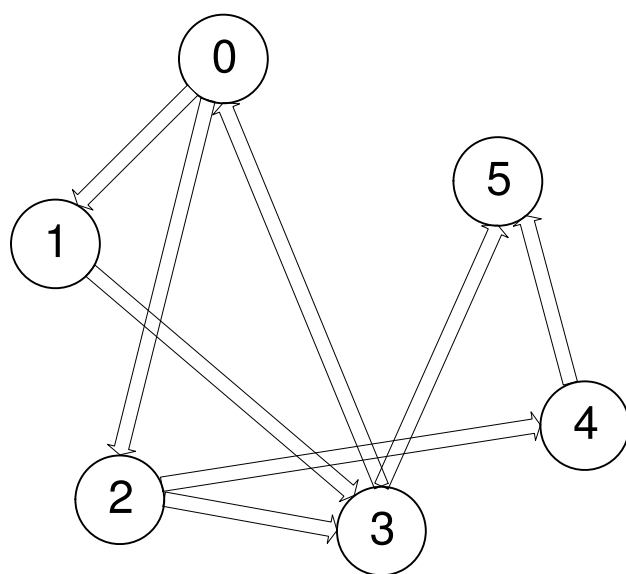
解法：

Tarjan算法定义了概念“强连通分支的根节点”。对有向图(G)中的节点(i) (范围为 $[0, n)$)，设置一个(low_i)值，表示从节点(i)出发进行DFS可以到达的所有节点中最小的节点标号。若节点(i)满足($low_i = i$)，则该节点为强连通分支的根节点。

Tarjan算法分为(2)个步骤：

((1))对图(G)中的所有节点进行深度优先搜索，计算每个遍历到的节点的(low)值，并将其压入栈(Stack)中；

((2))依次从栈(Stack)中取出每个节点，并放入初始为空的集合(S)中。对于刚刚进入集合的节点(i)，若该节点为强连通分支的根节点，则当前集合(S)中的所有节点属于一个强连通分支。将它们全部取出作为一个强连通分支，将集合(S)清空，然后继续从栈(Stack)中取出节点，重复该操作。



节点的low值分别为([0_0, 0_1, 0_2, 0_3, 4_4, 5_5]), 遍历顺序为([0, 1, 3, 2, 4, 5]), 依次压入栈后得到(Stack = [5, 4, 2, 3, 1, 0])。

((1))将节点(5)放入集合($S = [5]$), 节点($low_5 = 5$), 即节点(5)为强连通分支的根节点, 因此([5])为一个强连通分支, 将集合(S)清空;

((2))将节点(4)放入集合($S = [4]$), 节点($low_4 = 4$), 即节点(4)为强连通分支的根节点, 因此([4])为一个强连通分支, 将集合(S)清空;

((3))将节点(2)放入集合($S = [2]$), 节点($low_2 = 0$), 不做任何操作;

((4))将节点(3)放入集合($S = [2, 3]$), 节点($low_3 = 0$), 不做任何操作;

((5))将节点(1)放入集合($S = [2, 3, 1]$), 节点($low_1 = 0$), 不做任何操作;

((6))将节点(0)放入集合($S = [2, 3, 1, 0]$), 节点($low_0 = 0$), 即节点(0)为强连通分支的根节点, 因此([2, 3, 1, 0])为一个强连通分支, 将集合(S)清空;

((7))栈(Stack)为空, 算法结束。最终得到的强连通分支有(3)个, 分别为([5])、([4])、([2, 3, 1, 0]);

在有(N)个节点的有向图(G)上运行Tarjan算法的时间复杂度为($O(N)$)。

Wikipedia:

- https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

Gabow - Gabow算法

Gabow算法

问题：

用Gabow算法求有向图(G)的强连通分支。

解法：

Gabow算法是Tarjan算法一个变种，区别在于low值（详见）的处理方式，Gabow算法分为(2)个步骤：

((1))对图(G)中的所有节点进行深度优先搜索，计算每个遍历到的节点的(low)值，并将其压入栈(Stack)中；

((2))依次从栈(Stack)中取出每个节点，并放入初始为空的集合(S)中。对于刚刚进入集合的节点(i)，若该节点为强连通分支的根节点，则当前集合(S)中的所有节点属于一个强连通分支。将它们全部取出作为一个强连通分支，将集合(S)清空，然后继续从栈(Stack)中取出节点，重复该操作。

Tarjan算法在节点数量为(N)的有向图上的时间复杂度为($O(N)$)。

TwoSatisfiability 2-SAT问题

- [TwoSatisfiability 2-SAT问题](#)

TwoSatisfiability 2-SAT问题

Cut 割

- [Cut 割](#)

Cut 割

双向递增递减子序列

- [Bidirectional Increasing-Decreasing Subsequence](#) - 双向递增递减子序列
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Bidirectional Increasing-Decreasing Subsequence - 双向递增递减子序列

问题

递减子序列和递增子序列的概念相同，但渐变方向相反，递减子序列的元素之间依次递减。

在长度为 n 的序列 $s = [1, n]$ 中寻找元素 $s[i]$ ，使得 $s[1, i]$ 中的最长递增子序列和 $s[i, n]$ 中的最长递减子序列，它们的长度总和最大。

解法

设 $f(i)$ 是以 $s[i]$ 作为最右边元素的最长递增子序列的长度， $g(i)$ 是以 $s[i]$ 作为最左边元素的最长递减子序列的长度。

最后返回 $\max\{f(i) + g(i) - 1\}$ ($i \in [1, n]$)，即所有 $f(i) + g(i) - 1$ 中的最大值，之所以减去 1 是因为 $s[1, i]$ 最右边的元素和 $s[i, n]$ 最左边的元素是同一个元素，重复了因此长度减 1。该算法的时间复杂度是 $O(n^2)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

DoubleConnectedComponent 双联通分支

- [DoubleConnectedComponent 双联通分支](#)

DoubleConnectedComponent 双联通分支

LeastCommonAncestor 最近公共祖先

- [LeastCommonAncestor 最近公共祖先](#)

LeastCommonAncestor 最近公共祖先

RangeExtremumQuery 区域最值查询

- [RangeExtremumQuery](#) 区域最值查询

RangeExtremumQuery 区域最值查询

Section-5 FlowNetwork 第5节 网络流

EdmondsKarp EdmondsKarp算法

- [EdmondsKarp EdmondsKarp算法](#)

EdmondsKarp EdmondsKarp算法

PushAndRelabel 压入与重标记

- [PushAndRelabel 压入与重标记](#)

PushAndRelabel 压入与重标记

Dinic Dinic算法

- [Dinic Dinic算法](#)

Dinic Dinic算法

DistanceLabel 距离标号算法

- [DistanceLabel 距离标号算法](#)

DistanceLabel 距离标号算法

RelabelToFront 重标记与前移算法

- [RelabelToFront 重标记与前移算法](#)

RelabelToFront 重标记与前移算法

HighestLabelPreflowPush 最高标号预留与推进算法

- [HighestLabelPreflowPush 最高标号预留与推进算法](#)

HighestLabelPreflowPush 最高标号预留与推进算法

DistanceLabel-AdjacentListVersion 距离标号算法-邻接表优化版

- [DistanceLabel-AdjacentListVersion 距离标号算法-邻接表优化版](#)

DistanceLabel-AdjacentListVersion 距离标号算法-邻接表优化版

Summary-Maxflow 最大流算法小结

- [Summary-Maxflow 最大流算法小结](#)

Summary-Maxflow 最大流算法小结

MinimumCost-Maxflow 最小费用最大流

- [MinimumCost-Maxflow 最小费用最大流](#)

MinimumCost-Maxflow 最小费用最大流

MultipleSourceMultipleSink-Maxflow 多源点、多汇点的最大流

- [MultipleSourceMultipleSink-Maxflow 多源点、多汇点的最大流](#)

MultipleSourceMultipleSink-Maxflow 多源点、多汇点的最大流

Connectivity 连通度

- [Connectivity 连通度](#)

Connectivity 连通度

NoSourceNoSink-VolumeBoundedFlow 无源点、无汇点、容量有上下界的流网络

- [NoSourceNoSink-VolumeBoundedFlow 无源点、无汇点、容量有上下界的流网络](#)

NoSourceNoSink-VolumeBoundedFlow 无源点、无汇点、容量有上下界的流网络

VolumeBounded-Maxflow 容量有上下界的最大流

- [VolumeBounded-Maxflow](#) 容量有上下界的最大流

VolumeBounded-Maxflow 容量有上下界的最大流

VolumeBounded-Minflow 容量有上下界的最小流

- [VolumeBounded-Minflow 容量有上下界的最小流](#)

VolumeBounded-Minflow 容量有上下界的最小流

Section-6 BinaryMatch 第6节 二分匹配

Hungarian 匈牙利算法

- [Hungarian 匈牙利算法](#)

Hungarian 匈牙利算法

HopcroftKarp Hopcroft-Karp算法

- [HopcroftKarp Hopcroft-Karp算法](#)

HopcroftKarp Hopcroft-Karp算法

MatchToMaxflow 二分匹配转化为最大流

- [MatchToMaxflow](#)
- [二分匹配转化为最大流](#)

MatchToMaxflow

二分匹配转化为最大流

KuhnMunkres Kuhn-Munkres算法

- [KuhnMunkres](#) [Kuhn-Munkres算法](#)

KuhnMunkres Kuhn-Munkres算法

Introduction-Domination,Independent,Covering,Clique 介绍支配集、独立集、覆盖集和团

- [Introduction-Domination,Independent,Covering,Clique](#) 介绍支配集、独立集、覆盖集和团

Introduction-Domination,Independent,Covering,Clique 介绍支配集、独立集、覆盖集和团

WeightedCoveringAndIndependentSet 最小点权覆盖和最大点权独立集

- [WeightedCoveringAndIndependentSet](#)
- [最小点权覆盖和最大点权独立集](#)

WeightedCoveringAndIndependentSet

最小点权覆盖和最大点权独立集

MinimumDisjointPathCovering 最小不相交路径覆盖

- [MinimumDisjointPathCovering](#)
- [最小不相交路径覆盖](#)

MinimumDisjointPathCovering

最小不相交路径覆盖

MinimumJointPathCovering 最小可相交路径覆盖

- [MinimumJointPathCovering 最小可相交路径覆盖](#)

MinimumJointPathCovering 最小可相交路径覆盖

Coloring 染色问题

- [Coloring 染色问题](#)

Coloring 染色问题

Chapter-6 Calculation 第6章 计算

- [Chapter-6 Calculation](#)
- [第6章 计算](#)

Chapter-6 Calculation

第6章 计算

1. [LargeNumber](#) 大数字
2. [DecimalConversion](#) 进制转换
3. [Exponentiation](#) 幂运算

LargeNumber 大数字

- [LargeNumber - 大数字](#)
 - [问题](#)
 - [解法](#)
 - [取反](#)
 - [加法](#)
 - [减法](#)
 - [乘法](#)
 - [除法](#)
 - [源码](#)
 - [测试](#)

LargeNumber - 大数字

问题

计算两个数字 a 和 b 的加减乘除，这些数字非常大，无法用编程语言中内置的 `int64` 、 `float64` 等类型来存储。

解法

对每个数字设置符号、整数区、小数区三个部分，模拟小学数学中的加减乘除运算过程，得到结果。

取反

如果 $a \neq 0$ ， a 的负值为它自己，如果 $a = 0$ ， a 的负值为 $-a$ 。

加法

数字 $a + b$ ，有两种情况：

- (1) 相同符号：可以把数字部分直接进行相加；
- (2) 不同符号：比如 $12 + (-4)$ ，可以转化为减法 $12 - 4$ 来计算，因此只需要考虑符号相同的两个数字相加；

对于 $a + b = c$ ，从低位开始依次计算：

\$\$

```

c[i] = ( c[i] + a[i] + b[i] ) % 10 \
c[i+1] = c[i+1] + ( c[i] + a[i] + b[i] ) / 10

```

\$\$

下面演示 $-125.39 + (-91.70935) = -217.09935$:

	0	1	2	5	.	3	9	0	0	0
+	0	0	9	1	.	7	0	9	3	5
sum	0	0	0	0	.	0	9	9	3	5

(1) 最后四个小数位相加没有进位，可以直接相加，得 9935 ；

carry				1						
	0	1	2	5	.	3	9	0	0	0
+	0	0	9	1	.	7	0	9	3	5
sum	0	0	0	0	.	0	9	9	3	5

(2) 十分位（小数点后第一位） $3 + 7$ 进 1 位，得 0 ；

carry										
	0	1	2	5	.	3	9	0	0	0
+	0	0	9	1	.	7	0	9	3	5
sum	0	0	0	7	.	0	9	9	3	5

(3) 个位 $5 + 1 + 1$ （包含进位 1 ），得 7 ；

carry		1								
	0	1	2	5	.	3	9	0	0	0
+	0	0	9	1	.	7	0	9	3	5
sum	0	0	1	7	.	0	9	9	3	5

(4) 十位 $2 + 9$ 进 1 位, 得 1 ;

carry										
	0	1	2	5	.	3	9	0	0	0
+	0	0	9	1	.	7	0	9	3	5
sum	0	2	1	7	.	0	9	9	3	5

(5) 百位 $1 + 0 + 1$ (包含进位 1), 得 2 。两个数字的千位都为 0 , 且没有进位, 算法结束;

减法

数字 $a - b$, 有两种情况:

- (1) a 和 b 正负不同: 比如 $12 - (-4)$ 和 $-12 - 4$, 可以转化为加法 $12 + 4$ 和 $-12 + (-4)$ 来计算;
- (2) a 和 b 正负相同且 $a < b$: 比如 $1 - 14$ 可以转化为 $-(14 - 1)$;
- (3) a 和 b 正负相同且 $a \geq b$: 直接计算 $a - b$;

对于 $a - b = c$ (其中 $a \geq b$), 从高位开始依次计算:

\$\$

```
c[i] =
\begin{cases}
a[i] - b[i] & a[i] \geq b[i] \end{cases}
```

```

a[i] + 10 - b[i], c[i+1] = c[i+1] - 1 & a[i] < b[i]
\end{cases}

```

\$\$

下面演示 $125.39 - 91.70935 = 33.68065$:

carry

	0	1	2	5	.	3	9	0	0	0
-	0	0	9	1	.	7	0	9	3	5
<hr/>										
sub	0	1	0	0	.	0	0	0	0	0

(1) 百位 $1 - 0$, 得 1 ;

carry

	0	1	2	5	.	3	9	0	0	0
-	0	0	9	1	.	7	0	9	3	5
<hr/>										
sub	0	0	3	0	.	0	0	0	0	0

(2) 十位 $2 < 9$, 从百位借 1 有 $10 + 2 - 9$ 得 3 , 百位变为 0 ;

carry

	0	1	2	5	.	3	9	0	0	0
-	0	0	9	1	.	7	0	9	3	5
<hr/>										
sub	0	0	3	4	.	0	0	0	0	0

(3) 个位 $5 \geq 1$, 得 4 ;

carry						1				
	0	1	2	5	.	3	9	0	0	0
-	0	0	9	1	.	7	0	9	3	5
sub	0	0	3	3	.	6	0	0	0	0

(4) 十分位 $3 < 7$ ，从个位借 1 有 $3 + 10 - 7$ 得 6，个位变为 3；

carry										
	0	1	2	5	.	3	9	0	0	0
-	0	0	9	1	.	7	0	9	3	5
sub	0	0	3	3	.	6	9	0	0	0

(5) 百分位 $9 - 0$ 得 9；

carry							1			
	0	1	2	5	.	3	9	0	0	0
-	0	0	9	1	.	7	0	9	3	5
sub	0	0	3	3	.	6	8	1	0	0

(5) 千分位 $0 < 9$ ，从百分位借 1 有 $10 + 0 - 9$ 得 1，百分位变为 8；

carry									1	
	0	1	2	5	.	3	9	0	0	0
-	0	0	9	1	.	7	0	9	3	5
sub	0	0	3	3	.	6	8	0	7	0

(5) 万分位 $0 < 3$ ，从千分位借 1 有 $10 + 0 - 3$ 得 7，千分位变为 0；

carry									1	
	0	1	2	5	.	3	9	0	0	0
-	0	0	9	1	.	7	0	9	3	5
sub	0	0	3	3	.	6	8	0	6	5

(5) 亿分位 $0 < 5$ ，从万分位借 1 有 $10 + 0 - 5$ 得 5，万分位变为 6；

乘法

数字 $a \times b$ ，把数字部分直接进行相乘，最终正负相同则结果为正，正负不同则结果为负。

对于 $a \times b = c$ ，从低位开始依次计算：

\$\$

```
c[i] = ( c[i] + a[i] * b[i] ) % 10 \
c[i+1] = c[i+1] + ( c[i] + a[i] * b[i] ) / 10
```

\$\$

演示 $-125.39 * 91.70935 = -11499.4353965$ 我们会在后面补上

除法

数字 `a \div b` ，除法的模拟有些麻烦，我们会在后面补上

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

DecimalConversion 进制转换

- [Decimal Conversion - 进制转换](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Decimal Conversion - 进制转换

问题

将整数 x 从 a 进制转化为 b 进制 (其中 $2 \leq a, b \leq 32$)。

解法

(1) 模拟加法运算的过程将数字 $xa = [8, 3, 7, 1]_a$ 转换为 $y\{b\} = [8, 3, 7, 1]_{b\}$, 有 $y[i]\{b\} = x[i]\{a\} * a^i \bmod b^i$, $y[i+1]\{b\} = y[i+1]\{b\} + (x[i]\{a\} * a^i \div b^i)$ 。

(1) 模拟减法运算的过程将数字 $xa = [8, 3, 7, 1]_a$ 转换为 $y\{b\} = [8, 3, 7, 1]_{b\}$, 有 $y[i]\{b\} = x[i]\{a\} * a^i \bmod b^i$, $y[i+1]\{b\} = y[i+1]\{b\} + (x[i]\{a\} * a^i \div b^i)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

Exponentiation 幂运算

- [Exponentiation - 求幂运算](#)
 - [问题](#)
 - [解法](#)
 - [Wikipedia Exponentiation by squaring](#)
 - [源码](#)
 - [测试](#)

Exponentiation - 求幂运算

问题

求整数 x 的 n 次方的最低 m 位数字。

解法

循环的计算 $x \times x \dots \times x$ 共 n 次乘法，然后截取最低 m 位数字，该算法的时间复杂度为 $O(n)$ ，显然我们希望有更快的算法。

快速求幂算法基于下面的递归公式：

$$\begin{aligned}
 x^n = & \\
 \begin{cases} 1 & n == 0 \\ x & n == 1 \\ x * (x^2)^{(n-1)/2} & x \text{ 为奇数} \\ (x^2)^{n/2} & x \text{ 为偶数} \end{cases}
 \end{aligned}$$

可以写出递归函数来算出 x^n 。因为次方运算会使 x 迅速增大，导致计算机无法存储结果，因此每次计算次方后都可以对 m 取模，防止 x 增大到 `int32` 无法表示。

该算法还可以改写为非递归的二进制形式，性能更高。

Wikipedia Exponentiation by squaring

- 2k-ary算法
- Sliding-Window算法(滑动窗口算法)

- Montgomery's ladder技术(M的阶梯算法)
-

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

Chapter-7 CombinatorialMathematics 第7章 组合数学

- [Chapter-7 Combination Mathematics](#)
- [第7章 组合数学](#)

Chapter-7 Combination Mathematics

第7章 组合数学

1. [KnowledgePoint](#) 知识要点
2. [FullPermutation](#) 全排列
3. [UniqueFullPermutation](#) 唯一的全排列
4. [Combination](#) 组合
5. [DuplicableCombination](#) (元素)可重复的组合
6. [Subset](#) 子集
7. [UniqueSubset](#) 唯一的子集
8. [Permutation](#) 排列
9. [PermutationGroup](#) 置换群
0. [Catalan](#) 卡特兰数

KnowledgePoint 知识要点

- [Knowledge Point - 知识要点](#)

Knowledge Point - 知识要点

集合划分：

集合 (X) 的划分是 (X) 的非空子集的集合，使得每一个 (X) 的元素都只包含在这些子集的其中一个内。

等价的说，集合 (P) 是 (X) 的划分，如果：

((1)) (P) 的元素都是 (X) 的子集，且不是空集；

((2)) (P) 的元素的并集等于 (X) ；

((3)) (P) 的任意两个元素的交集为空集；

集合 (P) 中的元素也称为 (X) 的一个部分。例如 $(X = \{1, 2, 3, 4, 5, 6\})$ 的一个划分是 $(P = \{ \{1\}, \{2, 6\}, \{3, 4\}, \{5\} \})$ ，而 $(\{1\}), (\{2, 6\}), (\{3, 4\}), (\{5\})$ 都是 (X) 的一个部分。

加法原理：

集合 (X) 的元素数量等于 (X) 的所有部分的元素数量之和，即 $(\mid X \mid = \mid X_1 \mid + \mid X_2 \mid + \cdots + \mid X_n \mid)$ 。

乘法原理：

若集合 (X) 中的所有元素都是由两个数字组成的序列，即序偶 $((a, b))$ 。其中第一个元素 (a) 来自拥有 (p) 个元素的集合 (A) ，第二个元素 (b) 来自拥有 (q) 个元素的集合 (B) 。则集合 (X) 的元素数量为 $(\mid X \mid = p \times q)$ 。

减法原理：

设集合 (Y) 包含集合 (X) ，集合 (X) 在 (Y) 中的补集为 (X') ，则 $(\mid X \mid = \mid Y \mid - \mid X' \mid)$ 。

除法原理：

集合 (X) 被划分为 (p) 个部分，每个部分的元素数量都为 (q) ，则 $(\mid X \mid = p \times q)$ 。

阶乘：

```
[
n! =
\begin{cases}
1 & n = 0 \\
1 \times 2 \times 3 \times \cdots \times n & \text{for all } n > 0
\end{cases}
]
```

也可以写作：

```
[
n! =
\begin{cases}
1 & n = 0 \\
\prod_{k=1}^n k & \text{for all } n > 0
\end{cases}
]
```

阶乘的递归定义为：

```
[
n! =
\begin{cases}
1 & n = 0 \\
(n-1)! \times n & \text{for all } n > 0
\end{cases}
]
```

]

组合：

在拥有 (n) 个不同元素（没有两两相同的元素）的集合 (A) 中，任意选出 (m) 个元素 $((m \leq n)$ ， (m) 和 (n) 都是自然数，即正整数）组成另一个集合 (B) ，称 (B) 为 (A) 的一个子集。集合没有顺序的概念，对于集合 (A) 中的任意元素 $((\forall x \in A))$ ，都有 $(x \in B)$ ，同时集合 (B) 中的任意元素 $((\forall y \in B))$ ，都有 $(y \in A)$ ，则集合 (A) 和 (B) 是相同的。比如集合 $(s_1 = \{1, 2, 3\})$ 、 $(s_2 = \{3, 2, 1\})$ 是相同的两个集合。

从 (n) 个元素的集合中任意取出 (m) 个元素能够组成的不同集合的数量为：

$$C_m^n = \frac{n!}{m! \cdot (n-m)!}$$

在二项式定理 (https://en.wikipedia.org/wiki/Binomial_coefficient) 中，二项式幂 $((1+x)^n)$ 的多项式展开中 (x^k) （其中 $(k \in [0, n])$ ）的系数即为 (C_k^n) ，等同于组合学中从 (n) 个元素中选出 (k) 个元素组成集合的所有组合数量。

例如对于 $((x+1)^2 = 1x^0 + 2x^1 + x^2)$ ，从 (2) 个元素中选取 (1) 个的组合数量为 (2) ；选取 (2) 个的组合数量为 (1) 。

对于 $((x+1)^3 = 1x^0 + 3x^1 + 3x^2 + x^3)$ ，从 (3) 个元素中选取 (1) 个的组合数量为 (3) ；选取 (2) 个的组合数量为 (3) ；选取 (3) 个的组合数量为 (1) 。

排列：

从 (n) 个不同的元素（没有两两相同的元素）中任意取 (m) 个元素 $((m \leq n)$ ， (m) 和 (n) 都是自然数，即正整数）排成一列，得到排列 (s) 。排列 $(s_1 = [1, 2, 3])$ 、 $(s_2 = [3, 2, 1])$ 、 $(s_3$

$= [2, 3]$) 两两各不相同, 只有当两个排列长度相同, 且相同位置的元素也相同时, 才称这两个排列相同。

从 (n) 个元素中任意取出 (m) 个元素组成的所有排列的数量为:

$$P_m^n = \frac{n!}{(n-m)!}$$

也写作: $(A_m^n = \frac{n!}{(n-m)!})$, 维基百科中特别提到中国大陆教材中写做 (A_n^m) 。特别的当 $(m = n)$ 时, (P_m^n) 称为全排列, $(P_m^n = 1 \times 2 \times 3 \times \cdots \times n = n!)$ 。

数学符号表:

- https://en.wikipedia.org/wiki/List_of_mathematical_symbols

FullPermutation 全排列

- [Full Permutation - 全排列](#)
 - [问题](#)
 - [StackOverflow上关于全排列的问题](#)
 - [Steinhaus-Johnson-Trotter算法](#)
 - [LeetCode](#)

Full Permutation - 全排列

问题

求拥有(n)个不同元素的数组($A = [a_0, a_1, a_2, \dots, a_{n-1}]$)的所有全排列。

解法：

本文介绍Steinhaus-Johnson-Trotter算法。

初始时假设数组($A = []$)，其全排列只有(1)个，即($[]$)本身。

在初始状态的基础上增加新的元素，新的元素可以插入在原数组中的任意位置。例如对于数组($B = [1, 2, 3]$)，新元素(x)可以在(3)个元素中选择(4)个任意位置插入，得到(4)种情况：

```
[
[x, 1, 2, 3] \
[1, x, 2, 3] \
[1, 2, x, 3] \
[1, 2, 3, x]
]
```

((1))在初始状态($A = []$)的基础上增加新的元素(a_0)，新元素的位置是唯一的，得到($A = [a_0]$)。得到(1)个排列：

```
[
[a_0] \
]
```

((2))在第((1))轮的基础上增加新的元素(a_1)，新元素可以插入的位置有(2)个，得到的排列有(2)个：

```
[
[a_0, a_1] \
[a_1, a_0]
]
```

((3))在第((2))轮的基础上增加新的元素(a_2)，对于第((2))轮中的每个排列，新元素可以插入的位置都有(3)个，得到的排列共有($2 \times 3 = 6$)个：

```
[
[a_0, a_1, a_2] \
[a_0, a_2, a_1] \
[a_2, a_1, a_0] \
[a_1, a_0, a_2] \
[a_2, a_0, a_1] \
[a_1, a_2, a_0]
]
```

重复上述操作，即可得到长度为(n)的数组($A = [a_0, a_1, a_2, \dots, a_{n-1}]$)的全排列。该算法的时间复杂度为($O(n!)$)。

StackOverflow上关于全排列的问题

- <http://stackoverflow.com/questions/9878846/listing-all-permutations-of-a-given-set-of-values>

Steinhaus-Johnson-Trotter算法

- https://en.wikipedia.org/wiki/Steinhaus%E2%80%93Johnson%E2%80%93Trotter_algorithm

LeetCode

- [leetcode-46](#)
- [leetcode-46 source.hpp](#)

-
- [Upper Folder](#) - 上一级目录
 - [Source Code](#) - 源码
 - [Test Code](#) - 测试

UniqueFullPermutation 唯一的全排列

- [Unique Full Permutation - 唯一的全排列](#)
 - [问题](#)

Unique Full Permutation - 唯一的全排列

问题

求拥有(n)个不同元素的数组($A = [a_0, a_1, a_2, \dots, a_{n-1}]$)的唯一的全排列，其中数组(A)中存在重复的元素。

比如($A = [1, 2, 3_1, 3_2]$)，其全排列为：

```
[
[3_2, 3_1, 1, 2] \
[3_1, 3_2, 1, 2] \
[3_1, 1, 3_2, 2] \
[3_1, 1, 2, 3_2] \
[3_2, 1, 3_1, 2] \
[1, 3_2, 3_1, 2] \
[1, 3_1, 3_2, 2] \
[1, 3_1, 2, 3_2] \
[3_2, 1, 2, 3_1] \
[1, 3_2, 2, 3_1] \
[1, 2, 3_2, 3_1] \
[1, 2, 3_1, 3_2] \
[3_2, 3_1, 2, 1] \
[3_1, 3_2, 2, 1] \
[3_1, 2, 3_2, 1] \
[3_1, 2, 1, 3_2] \
[3_2, 2, 3_1, 1] \
[2, 3_2, 3_1, 1] \
[2, 3_1, 3_2, 1] \
[2, 3_1, 1, 3_2] \
[3_2, 2, 1, 3_1] \
[2, 3_2, 1, 3_1] \
[2, 1, 3_2, 3_1] \
[2, 1, 3_1, 3_2]
```

]

由于数组(A)中重复的元素,产生的全排列中也存在([1, 2, 3_1, 3_2] [1, 2, 3_2, 3_1])这样重复的排列,但实际上我们只需要一个([1, 2, 3, 3])。因此它的唯一的全排列为:

```
[
[3, 3, 1, 2] \
[3, 1, 3, 2] \
[3, 1, 2, 3] \
[1, 3, 3, 2] \
[1, 3, 2, 3] \
[1, 2, 3, 3] \
[3, 3, 2, 1] \
[3, 2, 3, 1] \
[3, 2, 1, 3] \
[2, 3, 3, 1] \
[2, 3, 1, 3] \
[2, 1, 3, 3]
]
```

解法:

在的基础上。

初始时假设数组(A = []), 其全排列只有(1)个, 即([])本身。

在初始状态的基础上增加新的元素, 新的元素可以插入在原数组中的任意位置。例如对于数组(B = [1, 2, 3]), 新元素(x)可以在(3)个元素中选择(4)个任意位置插入, 得到(4)种情况:

```
[
[x, 1, 2, 3] \
[1, x, 2, 3] \
[1, 2, x, 3] \
[1, 2, 3, x]
]
```

((1))在初始状态(A = [])的基础上增加新的元素(a_0), 新元素的位置是唯一的, 得到(A = [a_0])。得到(1)个排列:

```
[
[a_0] \
]
```

((2))在第((1))轮的基础上增加新的元素(a_1)，新元素可以插入的位置有(2)个，得到的排列有(2)个：

```
[
[a_0, a_1] \
[a_1, a_0]
]
```

((3))在第((2))轮的基础上增加新的元素(a_2)，对于第((2))轮中的每个排列，新元素可以插入的位置都有(3)个，得到的排列共有($2 \times 3 = 6$)个：

```
[
[a_0, a_1, a_2] \
[a_0, a_2, a_1] \
[a_2, a_1, a_0] \
[a_1, a_0, a_2] \
[a_2, a_0, a_1] \
[a_1, a_2, a_0]
]
```

重复上述操作，即可得到长度为(n)的数组($A = [a_0, a_1, a_2, \dots, a_{n-1}]$)的全排列。该算法的时间复杂度为($O(n!)$)。

Online Judge:

- [leetcode-47](#)
- [leetcode-47 source.hpp](#)

-
- [Upper Folder](#) - 上一级目录
 - [Source Code](#) - 源码
 - [Test Code](#) - 测试

Combination 组合

- [Combination - 组合](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Combination - 组合

问题

从拥有 n 个元素的集合 $A = \{a_0, a_1, a_2, \dots, a_{n-1}\}$ 中任意取 m 个元素 ($m \leq n$, m 和 n 都是自然数), 求所有组合。

解法

本文末尾列了很多关于组合算法的文献。本文介绍一种简单易记的算法。

设置数组($s = [s_0, s_1, s_2, \dots, s_{n-1}]$)表示对集合(A)的选择, 第(i)个数字($s_i = 1$)表示选择数字(a_i), ($s_i = 0$)表示不选择数字(a_i)。假设初始状态下从集合($A = \{a_0, a_1, a_2, \dots, a_{n-1}\}$)中取出(0)个元素组成组合, 即($s = [0_0, 0_1, 0_2, \dots, 0_{n-1}]$), 得到(1)个组合($\{\}$)。

(1) 从集合 $A = \{a_0, a_1, a_2, \dots, a_{n-1}\}$ 中取出 1 个元素作为组合。则数组 s 中只有一个元素为 1 , 其他全是 0 , 唯一的 1 在数组 s 中可以选择任意位置, 得到 $C_{1^n} = n$ 个组合:

```
[
\begin{array}{lcr}
[1_0, 0_1, 0_2, \dots, 0_{n-1}] & \& (1-1) \setminus \\
[0_0, 1_1, 0_2, \dots, 0_{n-1}] & \& (1-2) \setminus \\
[0_0, 0_1, 1_2, \dots, 0_{n-1}] & \& (1-3) \setminus \\
& \& \vdots \setminus \\
[0_0, 0_1, 0_2, \dots, 1_{n-1}] & \& (1-n) \\
\end{array}
]
```

((2))从集合A中取出2个元素作为组合, 可以看作是在第1轮操作数组s后得到的n个组合的基础上增加一个1。

对于第(1)轮中的数组 $((1-1) \quad [1_0, 0_1, 0_2, \dots, 0_{n-1}])$ 增加一个(1)后得到数组 $([1_0, 1_1, 0_2, \dots, 0_{n-1}])$ 。原本的 (1_0) 保持不变,新增的 (1_1) 可以选择后面等于 (0) 的 $(n-1)$ 个位置,生成 $(n-1)$ 个组合:

```
[
\begin{array}{lcr}
[1_0, 1_1, 0_2, 0_3, \dots, 0_{n-1}] && (2-1-1) \setminus \\
[1_0, 0_1, 1_2, 0_3, \dots, 0_{n-1}] && (2-1-2) \setminus \\
[1_0, 0_1, 0_2, 1_3, \dots, 0_{n-1}] && (2-1-3) \setminus \\
&& \& \dots \& \setminus \\
[1_0, 0_1, 0_2, 0_3, \dots, 1_{n-1}] && (2-1-(n-1)) \\
\end{array}
]
```

需要注意的是,新增的(1)必须在原数组中所有的(1)的后面。对于数组 $((1-2) \quad [0_0, 1_1, 0_2, \dots, 0_{n-1}])$,新增的(1)只能选择后面等于 (0) 的 $(n-1)$ 个位置,生成 $(n-2)$ 个组合:

```
[
\begin{array}{lcr}
[0_0, 1_1, 1_2, 0_3, \dots, 0_{n-1}] && (2-2-1) \setminus \\
[0_0, 1_1, 0_2, 1_3, \dots, 0_{n-1}] && (2-2-2) \setminus \\
&& \& \dots \& \setminus \\
[0_0, 1_1, 0_2, 0_3, \dots, 1_{n-1}] && (2-2-(n-2)) \\
\end{array}
]
```

如果不注意,让新增的(1)在原数组的任意的(1)的前面,则会产生重复的组合,仍然以数组 $((1-2) \quad [0_0, 1_1, 0_2, \dots, 0_{n-1}])$ 为例,如果新增的(1)可以选择任意等于 (0) 的位置,会生成 $(n-1)$ 个组合:

```
[
\begin{array}{lcr}
[1_0, 1_1, 0_2, 0_3, \dots, 0_{n-1}] && (2-2-0) \setminus \\
[0_0, 1_1, 1_2, 0_3, \dots, 0_{n-1}] && (2-2-1) \setminus \\
[0_0, 1_1, 0_2, 1_3, \dots, 0_{n-1}] && (2-2-2) \setminus \\
&& \& \dots \& \setminus \\
[0_0, 1_1, 0_2, 0_3, \dots, 1_{n-1}] && (2-2-(n-2)) \\
\end{array}
]
```

但其中 $(2-2-0) \quad [1_0, 1_1, 0_2, 0_3, \dots, 0_{n-1}]$ 与第(1)个数组产生的组合重复了。对第(1)轮中所有的数组重复该操作, 即可得到选取(2)个元素的所有组合, 共有 $(C_2^n = \frac{n \times (n-1)}{2})$ 个。

((3))从集合(A)中取出(3)个元素作为组合, 可以看作是在第((2))轮操作的 $((n-1)!)$ 个组合基础上增加一个(1), 对于之前的每个组合, 保持之前两个(1)不变, 新的(1)可以选择原数组中最后一个(1)之后的任意等于(0)的位置。注意新增的(1)不能比原数组中的任意的(1)更靠前, 必须在所有的(1)之后的位置进行选择。

重复上述的操作, 直到选取(m)个元素, 即可得到所有的组合, 算法结束。然后根据(s)的全排列生成集合(A)的所有组合即可。该算法时间复杂度为 $(C_m^n = \frac{n!}{m! \cdot (n-m)!})$ 。

StackOverflow上关于组合产生算法的问题:

- <http://stackoverflow.com/questions/127704/algorithm-to-return-all-combinations-of-k-elements-from-n>

二项式系数:

- https://en.wikipedia.org/wiki/Binomial_coefficient

Chase's Twiddle - Algorithm 382: Combinations of M out of N Objects:

- <http://dl.acm.org/citation.cfm?id=362502>
- <http://www.netlib.no/netlib/toms/382>

Buckles - Algorithm 515: Generation of a Vector from the Lexicographical Index:

- <http://dl.acm.org/citation.cfm?id=355739>
- https://www.researchgate.net/profile/Bill_Buckles/publication/220492658_Algorithm_515_Generation_of_a_Vector_from_the_Lexicographical_Index_G6/links/5716d7ad08ae497c1a5706ec.pdf

Remark on algorithm 515: Generation of a vector from the lexicographical index combinations:

- <http://dl.acm.org/citation.cfm?id=1236470>

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

DuplicableCombination (元素) 可重复的组合

- [Duplicable Combination](#)
- [\(元素\) 重复的组合](#)
- [Upper Folder](#) - 上一级目录
- [Source Code](#) - 源码
- [Test Code](#) - 测试

Duplicable Combination

(元素) 重复的组合

问题:

在有 (n) 个互不相同元素的集合 $(A = \{a_0, a_1, a_2, \dots, a_{n-1}\})$ 中, 任意取 (m) 个元素 $((m \leq n), (m)$ 和 (n) 都是自然数, 并且同一个元素可以重复使用)的所有组合。例如对于集合 $(A = \{1, 2, 3\})$, 从中取出 (3) 个元素可重复的所有组合为:

```
[
[1, 1, 1] \
[1, 1, 2] \
[1, 2, 1] \
[2, 1, 1] \
[1, 2, 2] \
[2, 1, 2] \
[2, 2, 1] \
[2, 2, 2] \
[2, 2, 3] \
[2, 3, 2] \
[3, 2, 2] \
[2, 3, 3] \
[3, 2, 3] \
[3, 3, 2] \
[3, 3, 3] \
[1, 1, 3] \
[1, 3, 1] \
[3, 1, 1] \
[1, 3, 3] \
```

```
[3, 1, 3] \
[3, 3, 1] \
[1, 2, 3] \
[1, 3, 2] \
[3, 2, 1] \
[3, 1, 2] \
[2, 1, 3] \
[2, 3, 1]
]
```

解法：

[Recursion](#)可以解决这个问题。所求的组合是长度为(m)的数组(S)，其中每个元素(i)可以选择的值为集合(A)中的任意一个元素。因此我们只需要递归的对每个元素i选择一个值，然后递归下去对元素i+1设置一个值，直到将数组中所有元素都选择了一个值，即可得到一个组合。

所求组合的长度为(m)，集合(A)的成员有(n)个，该算法的时间复杂度($O(m^n)$)。

Online Judge：

- [leetcode-40](#)
- [leetcode-40 source.hpp](#)

Subset 子集

- [Subset - 子集](#)
 - [问题](#)

Subset - 子集

问题

问题：

求拥有(n)个元素的集合($A = \{a_0, a_1, a_2, \dots, a_{n-1}\}$)的所有子集。

解法：

拥有(n)个元素的集合(A)的所有子集，可以看作从集合(A)中取出(0)个元素的所有组合、取出(1)个元素的所有组合、...取出(n-1)个元素的所有组合、取出(n)个元素的所有组合，这些所有组合即为所求。

具体从n个元素的集合中取出m个元素的所有组合，可以通过中的算法求出。因此子集问题只需要遍历所有元素个数即可。

该算法的时间复杂度为($O(2^n)$)。

-
- [Upper Folder - 上一级目录](#)
 - [Source Code - 源码](#)
 - [Test Code - 测试](#)

UniqueSubset 唯一的子集

- [Unique Subset - 唯一的子集](#)
 - [问题](#)
 - [解法](#)

Unique Subset - 唯一的子集

问题

求拥有(n)个元素的集合($A = \{a_0, a_1, a_2, \dots, a_{n-1}\}$)中的不重复的所有子集。

解法

拥有(n)个元素的集合(A)的所有子集，可以看作从集合(A)中取出(0)个元素的所有组合、取出(1)个元素的所有组合、...取出(n-1)个元素的所有组合、取出(n)个元素的所有组合，这些所有组合即为所求。

具体从n个元素的集合中取出m个元素的所有组合，可以通过中的算法求出。因此子集问题只需要遍历所有元素个数即可。

该算法的时间复杂度为($C_m^n = \frac{n!}{m! \cdot (n-m)!}$)。

-
- [Upper Folder - 上一级目录](#)
 - [Source Code - 源码](#)
 - [Test Code - 测试](#)

Permutation 排列

- [Permutation - 排列](#)
 - [问题](#)

Permutation - 排列

问题

求(n)个不同元素($A = [a_0, a_1, a_2, \dots, a_{n-1}]$)中任意取(m)个元素 ($m \leq n$), (m)和(n)都是自然数)的所有排列。

解法:

在<Full Permutation>和<Combination>的基础上可知, 从拥有(n)个元素的(A)中任意选取(m)个元素, 得到的所有组合的集合为(P), (P)中的每个元素都是(A)的一种组合, 且任意两个元素不相同。对(P)中的每个元素进行全排列, 得到的排列即为所求。

比如对于($A = [1, 2, 3, 4, 5]$), 从中取出(3)个元素。其所有组合为: ($[1, 2, 3]$)、($[1, 2, 4]$)、($[1, 2, 5]$)、($[1, 3, 4]$)、($[1, 3, 5]$)、($[1, 4, 5]$)、($[2, 3, 4]$)、($[2, 3, 5]$)、($[3, 4, 5]$)。

对其中的每个组合都进行全排列。其中($[1, 2, 3]$)的全排列为: ($[1, 2, 3]$)、($[2, 1, 3]$)、($[2, 3, 1]$)、($[3, 2, 1]$)、($[3, 1, 2]$)、($[1, 3, 2]$)。类似的对其他组合也进行全排列, 得到的所有排列即为从($A = [1, 2, 3, 4, 5]$)中取出(3)个元素得到的所有排列。

该算法的时间复杂度为($P_m^n = \frac{n!}{(n-m)!}$)。

-
- [Upper Folder - 上一级目录](#)
 - [Source Code - 源码](#)
 - [Test Code - 测试](#)

PermutationGroup 置换群

- [Permutation Group - 置换群](#)
 - [问题](#)

Permutation Group - 置换群

问题

长度为 (n) 的序列($s = [x_0, x_1, x_2, \dots, x_{n-1}]$)上有 (n) 个数字, 每个数字各不相同, 且任意的数字都满足($\forall x_i \in [0, n-1]$)。例如($s = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$)就是这样一个长度为 (10) 的序列, 拥有 (10) 个各不相同的数字, 且每个数字都满足($i \in [0, 9]$)。

给出长度相同的序列($t = [y_0, y_1, y_2, \dots, y_{n-1}]$), 与序列 (s) 满足相同的条件 (有 (n) 个数字, 每个数字各不相同, 且任意的数字都满足($\forall y_i \in [0, n-1]$))。例如

```
[t = [3, 4, 2, 6, 1, 7, 0, 5, 9, 8]]
```

我们将序列 (t) 作为序列 (s) 的置换准则, 置换操作可以令($s[t[i]] = s[i]$), 其中($i \in [0, n-1]$)。将序列 (s) 中的所有元素按照序列 (t) 中的下标进行一次置换, 称为一次置换操作。例如

```
[s = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
```

经过序列 (t) 的一次置换后, 变成

```
[6, 4, 2, 0, 1, 7, 3, 5, 9, 8]]
```

求长度为 (n) 的序列 (s) 在置换原则 t 下, 经过 (k) 次置换操作后的元素排列情况。

解法:

暴力 (k) 次循环时间复杂度过高。我们来仔细考察上面例子中的序列 (s) 及其置换准则 (t) :

```
[
s = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] \
t = [3, 4, 2, 6, 1, 7, 0, 5, 9, 8]
]
```

((1))对于第 (0) 个元素, 第 (1) 次置换后($s[0] = 6$);

```
[ s_1 = [6, 4, 2, 0, 1, 7, 3, 5, 9, 8] ]
```

((2))对于第 (0) 个元素, 第 (2) 次置换后($s[0] = 3$);

```
[ s_2 = [3, 1, 2, 6, 4, 5, 0, 7, 8, 9] ]
```

((3))对于第(0)个元素,第(3)次置换后($s[0] = 0$);

```
[ s_2 = [0, 4, 2, 3, 1, 7, 6, 5, 9, 8] ]
```

((4))对于第(0)个元素,第(4)次置换后($s[0] = 6$);

```
[ s_2 = [6, 1, 2, 0, 4, 5, 3, 7, 8, 9] ]
```

我们可以看出第(4)次置换和第(1)置换后($s[0]$)的结果相同,这说明置换操作是有周期性的,第0个元素($s[0]$)的周期为3,即($s[0]\{i+3\} = s[0]\{i\}$),其中($i \in [0, 6]$)。不同元素的周期是不同的,比如($s[2]$)的周期为1 ($(s[2]\{i+1\} = s[2]\{i\})$,其中($i \in [0, 8]$)),因为($s[2] \equiv 2$)。

经过观察发现任意元素($s[i]$)都拥有一个循环周期,因此只要确定每个元素的周期,就可以避免暴力循环,直接求出k次置换操作后的序列s。该算法的时间复杂度为($O(n)$)。

- [Upper Folder](#) - 上一级目录
- [Source Code](#) - 源码
- [Test Code](#) - 测试

Catalan 卡特兰数

- [Catalan - 卡特兰数](#)
 - [问题](#)
 - [解法](#)
- [include](#)
- [include](#)
- [include](#)
 - [源码](#)
 - [测试](#)

Catalan - 卡特兰数

问题

卡特兰数（又称卡塔兰数）是组合数学中常见的数列，在计算凸多边形面积划分和棋盘路径、以及进出栈方法数中具有很多应用。

卡特兰数 $C_{\{n\}}$ 满足以下递推关系：

\$\$

$$f(n) = \frac{1}{n+1} \left(\begin{cases} 2 \times n \times n \\ n \end{cases} \right) \text{ 其中 } (n \geq 0)$$

\$\$

卡特兰数可以解决的问题：

计算n+2条边的凸多边形中划分三角形的个数

在此模型中，选取任意的两条边为基点，在剩余的n条边中，可知是两部分的卷积公式 $C_n = \sum (C_i * C_{n-i}) \quad 0 < i < n$

计算网格中的路径方案数：

在n*n的网格中，求从左下角到右上角的路径的方案数，
要求不能穿过对角线

这个问题可以转换成第三中模型

进出栈的问题：

设有n个1和n个-1随机组合，在其中添加括号，是的每个括号中的值都不为负数，也就是一类dyck word数的计算

dyck word数：是一个有n个X和n个Y组成的字串，且所有部分的字串满足x的个数不小于y的个数，一下为5中情况(n=3)

XXXXYY XYXXYY XYXYXY XXYYXY XXYYXY

将上述X换成左括号，Y换成右括号，Cn表示组合式算法个数C3=5.

求出 n 的卡特兰数 f(n)

解法

include

include

include

```
using namespace std;
```

```
void catalan(int*a,int b) //求卡特兰数
```

```
{
```

```
int i,j,len,carry,temp;
```

```
a[1][0]=b[1]=1;
```

```
len=1;
```

```
for(i=2;i<=100;i++)
```

```
{
```

```
for(j=0;j<len;j++) //除法
```

```
{
```

```
temp=carry*10+a[i][j];
```

```
a[i][j]=temp/(i+1);
```

```
carry=temp%(i+1);
```

```
}
```

```
while(!a[i][len-1]) //高位零处理
```

```
len--;
```

```
b[i]=len;
```

```
}
```

```
}
```

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

Chapter-8 NumberTheory 第8章 数论

- [Chapter-8 NumberTheory](#)
- [第8章 数论](#)

Chapter -8 NumberTheory

第8章 数论

1. [Sieve](#) 筛选算法
2. [Euclid](#) 欧几里得
3. [EuclidExtension](#) 欧几里得扩展
4. [ModularLinearEquation](#) 模线性方程
5. [ChineseRemainerTheorem](#) 中国剩余定理
6. [ModularExponentiation](#) 模幂运算

Sieve 筛选算法

- [Sieve - 筛选算法](#)
 - [问题](#)
 - [解法1](#)
 - [源码](#)
 - [测试](#)
- [include "general_head.h"](#)

Sieve - 筛选算法

问题

素数是除了 1 和它自身没有其他数能够整除的正整数，最小的素数是 2。而不符合该特性的正整数是合数。素数是数论学科中的基础概念，关于素数的最为著名的问题就是哥德巴赫猜想。

判断 1 - n 中见的任意一个整数是不是素数。常见的素数有 2, 3, 5, 7, 11, 13, 17, 19, 23 等等。

解法1

基本判断

判断一个正整数 x 是否为素数，按照素数的定理，只要不断计算 $result = x \times i$ ($1 \leq i \leq x$)，累加得到result

埃拉托斯特尼 (Eratosthenes) 筛选法

设置一个数组 $s = [1...n]$ ，每个位置 $s[i]$ 为 true 表示它是素数，否则为合数。

```
(1) 从 2 开始，2 为素数，以 2 为筛子，留下2，删去所有2的倍数
//2之后第一个未被删除的数是3，则3为素数，以3为筛子，留下3，删去所有3的倍数
//3之后第一个未被删除的数是5，则5为素数，以5为筛子，留下5，删除所有5的倍数
//以此类推...
//当求解的数字范围较大时可以使用long long这样的数据类型
```



这样每次查找单词时，按照前缀从根节点开始向下匹配每个孩子节点的字符即可。前缀树查找一个长度为 n 的单词的时间复杂度为 $O(n)$ 。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```

include "general_head.h"

```
void sieve1(int n, int prime)
{
    //求从2到n-1中的所有素数，返回prime数组，prime[i]指代数字i是否为素数
    memset(prime, 1, MAX sizeof(int));
    for(int i = 2; i < n; ++ i)
        if(prime[i])
            for(int j = i * 2; j < n; j += i)
                prime[j] = 0;
}
```

//2)对埃拉托斯特尼筛选法的优化

//因为素数中只有2是偶数，故可以只筛选奇数，将循环的速度加快

```
void sieve2(int n, int prime)
{
    memset(prime, 1, MAX sizeof(int));
    //prime数组中偶数位置仍然是1，输出结果时需要注意
    for(int i = 3; i < n; i += 2)
        if(prime[i])
            for(int j = i * i; j < n; j += i)
                prime[j] = 0;
}
```

//3)快速筛选法

```
void sieve3(int n, int prime)
{
    //prime数组的使用方法与之前不同，prime[i]指代一个素数数字的值，下标从0到n-1
    int not_prime[MAX], cnt(0);
    memset(not_prime, 0, MAX sizeof(int));
    memset(prime, 0, MAX sizeof(int));
    for(int i = 2; i < n; ++ i){
        if(!not_prime[i])
            prime[cnt ++] = i;
    }
```



```
for(int j = 0; j < cnt and i prime[j] < n; ++ j){  
    not_prime[i * prime[j]] = 1;  
    if(!(i % prime[j]))  
        break;  
}  
}  
}
```

Euclid 欧几里得

EuclidExtension 欧几里得扩展

ModularLinearEquation 模线性方程

ChineseRemainerTheorem 中国剩余定理

ModularExponentiation 模幂运算

- [ModularExponentiation 模幂运算](#)

ModularExponentiation 模幂运算

Chapter-9 LinearAlgebra 第9章 线性代数

- [Chapter-9 LinearAlgebra 第9章 线性代数](#)

Chapter-9 LinearAlgebra 第9章 线性代数

Section-1 Matrix 第1节 矩阵

- [Section-1 Matrix 第1节 矩阵](#)

Section-1 Matrix 第1节 矩阵

Strassen Strassen算法

- [Strassen](#)
- [Strassen算法](#)

Strassen

Strassen算法

GaussElimination 高斯消元法

- [GaussElimination](#)
- [高斯消元法](#)

GaussElimination

高斯消元法

LUP LUP分解

- [LUP](#)
- [LUP分解](#)

LUP

LUP分解

InverseMatrix 矩阵求逆

- [InverseMatrix](#)
- [矩阵求逆](#)

InverseMatrix

矩阵求逆

Section-2 LinearProgramming 第2节 线性规划

- [Section-2 LinearProgramming 第2节 线性规划](#)

Section-2 LinearProgramming 第2节 线性规划

Simplex 单纯形算法

- [Simplex 单纯形算法](#)

Simplex 单纯形算法

Dinkelback Dinkelback算法

- [Dinkelback Dinkelback算法](#)

Dinkelback Dinkelback算法

Chapter-10 AnalyticGeometry 第10章 解析几何

- [Chapter-10 AnalyticGeometry 第10章 解析几何](#)

Chapter-10 AnalyticGeometry 第10章 解析几何

Section-1 Polygon 第1节 多边形

- [Section-1 Polygon 第1节 多边形](#)

Section-1 Polygon 第1节 多边形

Cross 叉积

- [Cross 叉积](#)

Cross 叉积

SegmentIntersection 线段相交

- [SegmentIntersection 线段相交](#)

SegmentIntersection 线段相交

PointInConvexPolygon 点在多边形内

- [PointInConvexPolygon 点在多边形内](#)

PointInConvexPolygon 点在多边形内

Sweeping 扫除算法

- [Sweeping 扫除算法](#)

Sweeping 扫除算法

ConvexPolygonArea 凸多边形面积

- [ConvexPolygonArea](#) 凸多边形面积

ConvexPolygonArea 凸多边形面积

ConvexPolygonGravityCenter 凸多边形重心

- [ConvexPolygonGravityCenter 凸多边形重心](#)

ConvexPolygonGravityCenter 凸多边形重心

RayDistinguish 射线判别

- [RayDistinguish 射线判别](#)

RayDistinguish 射线判别

RotatingCalipers 旋转卡壳

- [RotatingCalipers 旋转卡壳](#)

RotatingCalipers 旋转卡壳

Section-2 ConvexHull 第2节 凸包

- [Section-2 ConvexHull 第2节 凸包](#)

Section-2 ConvexHull 第2节 凸包

NearestNeighbor 最近点对

- [NearestNeighbor 最近点对](#)

NearestNeighbor 最近点对

GrahamScan Graham扫描算法

- [GrahamScan Graham扫描算法](#)

GrahamScan Graham扫描算法

QuickConvexHull 快速凸包算法

- [QuickConvexHull 快速凸包算法](#)

QuickConvexHull 快速凸包算法

Chapter-11 TextMatch 第11章 文本匹配

- [Chapter-11 TextMatch](#)
- [第11章 文本匹配](#)

Chapter -11 TextMatch

第11章 文本匹配

1. [SimpleMatch](#) 简单匹配
2. [KMPMatch](#) - KMP匹配
3. [ACAutomation](#) - AC自动机

SimpleMatch 简单匹配

- [Simple Match - 简单匹配](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

Simple Match - 简单匹配

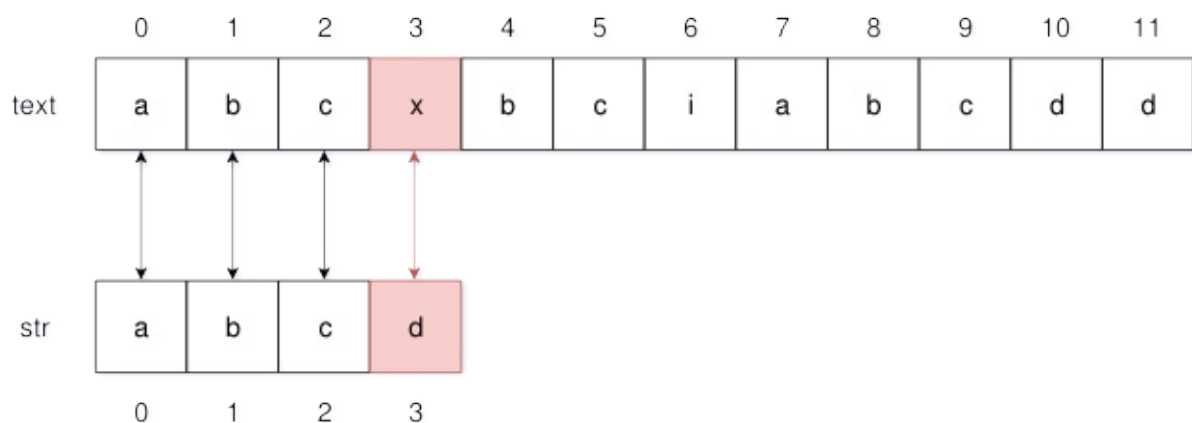
问题

在文本 `text` 中查找字符串 `str` 的位置（比如文本编辑器 VIM 中查找某个字符串的场景，设 `text` 长度为 `n`，`str` 长度为 `m`，该场景满足 $n > m$ ）。

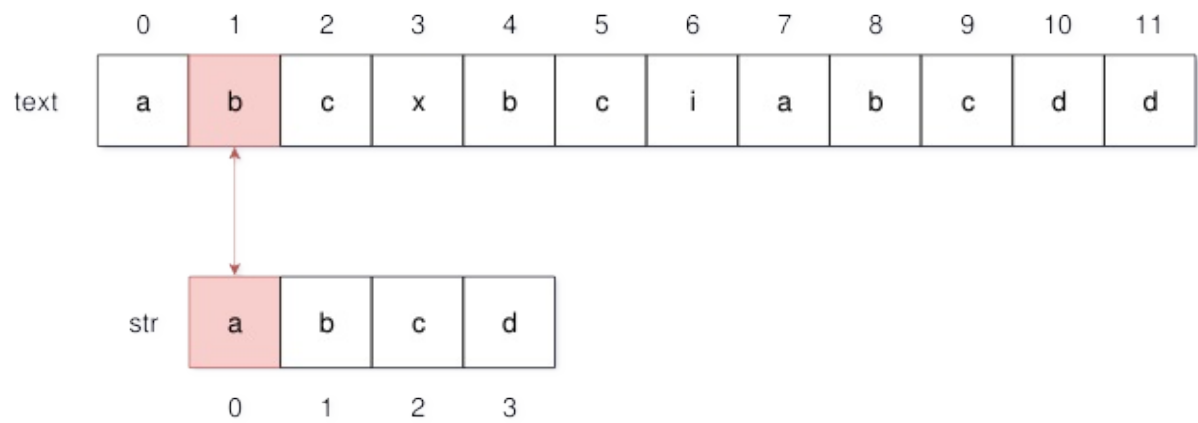
解法

假设字符数组下标从 1 开始。简单匹配的方法是对于 `text[1 .. n]`，从第 1 个字符开始，依次比较 `text[i+1 .. i+m]` 是否与 `str[1 .. m]` 相等。若相等则 `text[1 .. n]` 中匹配 `str[1 .. m]` 的部分为 `text[i+1 .. i+m]`，否则继续从下一个位置 `i+1` 匹配。

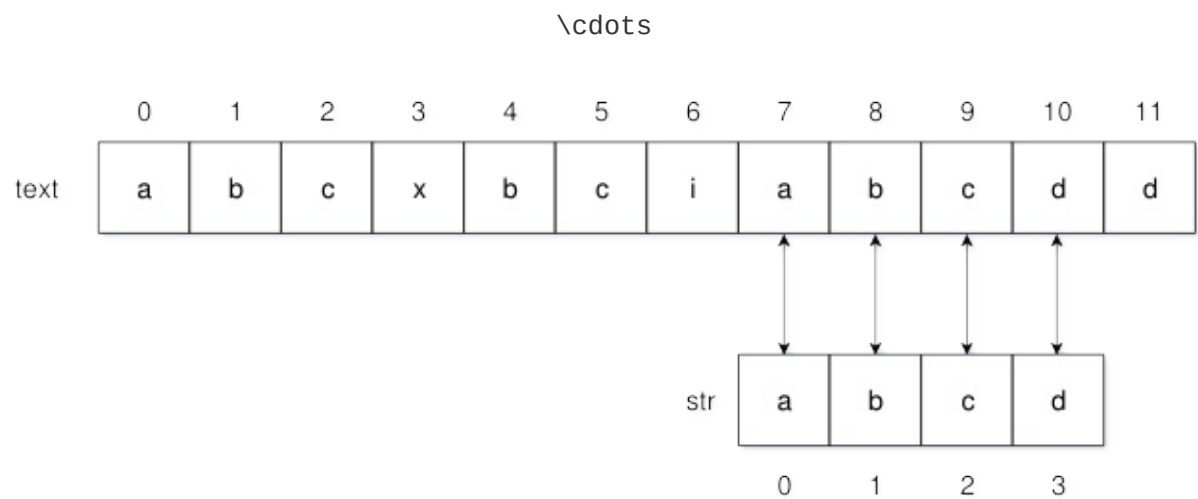
实际大部分编程语言中字符数组的下标都从 0 开始。下面演示一个匹配的过程：



(1) 从 `text` 第 0 个字符开始匹配，有 `text[0 .. 2] = str[0 .. 2]`，`text[3] != str[3]`，继续匹配下一个字符；



(2) 从 text 第 1 个字符开始匹配, 有 `text[1] \neq str[0]` , 继续匹配下一个字符;



(3) 从 text 第 7 个字符开始匹配, `text[7 \cdots 10] = str[0 \cdots 3]` , 匹配成功, 算法结束;

显然, 对于 text 中的每个位置 `i` , 都需要进行一次匹配, 而每次匹配的平均时间复杂度为 str 的长度 `m` 。因此该算法的时间复杂度为 $O(m \times n)$, 实现简单但性能差。

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```


KMPMatch KMP匹配算法

- [KMP\(Knuth Morris Pratt\) Match - KMP匹配](#)
 - [问题](#)
 - [解法](#)
 - [源码](#)
 - [测试](#)

KMP(Knuth Morris Pratt) Match - KMP匹配

问题

在文本 `text` 中查找字符串 `str` 的位置（设 `text` 长度为 n ，`str` 长度为 m ，该场景满足 $n > m$ ）。

解法

KMP算法的性能为 $O(n)$ ，比SimpleMatch高很多。在 `text = abcxbciaabcab` 搜索 `str = abcab` 的过程中，其实仔细观察一下会发现，没有必要在每次失败的时候，都重新从 `str` 的起始处开始匹配，我们可以跳过一部分。

对于下面这个匹配：



(1) 从 `text` 第 0 个字符开始匹配，有 `text[0 \cdots 2] \neq str[0 \cdots 2]`，`text[3] \neq str[3]`；

(2) 这次我们不希望从 `text` 的第 1 个字符开始匹配，因为这样需要重复比较 `str[0 \cdots 2]` 这 3 个字符，导致时间复杂度成为 $O(n \times m)$ 。

这里我们定义两个概念：前缀、后缀。设字符串 `s` 的前缀 `prefix`、后缀 `suffix` 分别表示该字符串的真子集（即 `prefix \neq s` 且 `suffix \neq s`），并且，前缀是从头开始的连续字符串子集，后缀是从尾开始的连续字符串子集，包括空字符串（空集）。比如字符串 `s = abcab`，其前缀为 `prefix = [\emptyset, a, ab, abc, abca]`，后缀为 `suffix = [\emptyset, b, ab, cab, bcab]`（注意前缀和后缀中不能有字符串 `s = abcab` 本身，必须是真子集）。

在前缀、后缀的概念下，对于长度为 m 的字符串 `str` 的任意一个子字符串 `str[0 \cdots`

$i]$, 我们都可以求出它的前缀和后缀。例如对于字符串 $str = abcab$, 可以得到:

(1) 子字符串 $str[0 \dots 0] = a$ 的前缀为 $prefix = [\text{\texttt{\textbackslash emptyset}}]$, 后缀为 $suffix = [\text{\texttt{\textbackslash emptyset}}]$;

(2) 子字符串 $str[0 \dots 1] = ab$ 的前缀为 $prefix = [\text{\texttt{\textbackslash emptyset}}, a]$, 后缀为 $suffix = [\text{\texttt{\textbackslash emptyset}}, b]$;

(3) 子字符串 $str[0 \dots 2] = abc$ 的前缀为 $prefix = [\text{\texttt{\textbackslash emptyset}}, a, ab]$, 后缀为 $suffix = [\text{\texttt{\textbackslash emptyset}}, c, bc]$;

(4) 子字符串 $str[0 \dots 3] = abca$ 的前缀为 $prefix = [\text{\texttt{\textbackslash emptyset}}, a, ab, abc]$, 后缀为 $suffix = [\text{\texttt{\textbackslash emptyset}}, a, ca, bca]$;

(5) 子字符串 $str[0 \dots 4] = abcab$ 的前缀为 $prefix = [\text{\texttt{\textbackslash emptyset}}, a, ab, abc, abca]$, 后缀为 $suffix = [\text{\texttt{\textbackslash emptyset}}, b, ab, cab, bcab]$;

设字符串 str 的每个字符 $str[i]$ 有个回文长度 $len[i]$ 。对于子字符串 $str[i]$ 的前缀、后缀, 找到其中相同的前缀和后缀 $same$ (如果存在多个, 选最长的), 则 $len[i] = length_{\{same\}}$ 。

(1) 子字符串 $str[0 \dots 0]$, 其相同的最长的前缀和后缀为 $\text{\texttt{\textbackslash emptyset}}$, 因此 $len[0] = 0$;

(2) 子字符串 $str[0 \dots 1]$, 其相同的最长的前缀和后缀为 $\text{\texttt{\textbackslash emptyset}}$, 因此 $len[1] = 0$;

(3) 子字符串 $str[0 \dots 2]$, 其相同的最长的前缀和后缀为 $\text{\texttt{\textbackslash emptyset}}$, 因此 $len[2] = 0$;

(4) 子字符串 $str[0 \dots 3]$, 其相同的最长的前缀和后缀为 a , 因此 $len[3] = 1$;

(5) 子字符串 $str[0 \dots 4]$, 其相同的最长的前缀和后缀为 ab , 因此 $len[4] = 2$;

最终得到的回文长度为 $len = [0, 0, 0, 1, 2]$ 。回文长度代表的意义是子字符串中从头开始、和从尾开始的两部分字符串的最长相同部分的长度。

回到开始, 当匹配 $text[0 \dots 4] \neq str[0 \dots 4]$ 时, 其中 $text[0 \dots 2] \{abc\} \neq str[0 \dots 2] \{abc\}$, 而 $text[3] \{x\} \neq str[3] \{a\}$ 。这时以最后一个成功匹配的字符 $str[2]$ 为准, 我们知道 $len[2] = 0$, 即 $str[2]$ 这个字符存在最大长度为2的前缀和后缀, 即 $str[0 \dots 1] \{ab\}$ 和 $str[3 \dots 4] \{ab\}$ 。这时可以算出 移动长度 = 已经匹配的字符数量 - 最后一位匹配字符的匹配长度值 , 即 $1 = 3 - 2$ 。

在上面这个例子中，该公式表达了，已经匹配的字符串 $\text{str}[0 \dots 2] = \text{abc}$ ，它的前缀 $\text{str}[0 \dots 1] = \text{ab}$ ，和它后面没有被匹配的后缀 $\text{str}[3 \dots 4] = \text{ab}$ 是相同的。利用这个特点，下一次匹配我们希望可以直接让前缀 ab 和

源码

```
import, lang:"c_cpp"
```

测试

```
import, lang:"c_cpp"
```



ACAutomation AC自动机

- [AC\(Aho Corasick\) Automation - AC自动机](#)
 - [问题](#)
 - [解法](#)

AC(Aho Corasick) Automation - AC自动机

问题

在文本 `text` 中查找 `k` 个字符串 `str` 出现的所有位置，其中 `text` 长度为 `n`，`str` 长度为 `m`。

如果使用SimpleMatch或KMPMatch，我们需要对每一个字符串 `str[i]`（其中 $0 \leq i < k$ ）都执行一遍匹配算法，对于长度为 `n` 的文本 `text`，一次匹配算法的时间复杂度最小为 $O(n)$ （KMPMatch），那么整个过程的时间复杂度为 $O(n \times k)$ 。

AC自动机算法可以在遍历一次文本 `text` 中找出 `k` 个字符串的所有位置，时间复杂度接近 $O(n)$ 。

解法

Chapter-12 GameTheory 第12章 博弈论

- [Chapter-12 GameTheory](#)
- [第12章 博弈论](#)

Chapter -12 GameTheory

第12章 博弈论

BashGame 巴什博弈

- [BashGame 巴什博弈](#)

BashGame 巴什博弈

WythoffGame 威佐夫博弈

- [WythoffGame 威佐夫博弈](#)

WythoffGame 威佐夫博弈

NimGame 尼姆博弈

- [NimGame 尼姆博弈](#)

NimGame 尼姆博弈

SpragueGrundy SG函数

- [SpragueGrundy SG函数](#)

SpragueGrundy SG函数
