目录

第一章

1.递推公式求解

(1)母函数法 —— 要用到级数

(2)特征根法

一些补充说明：其他老师 ppt 上写线性非齐次递归方程（不做要求），不做特解就好
使用特征根解法需要判断递推方程是线性还是非线性，**写出特征方程**，对特征方程进行求解，根据特征方程的解是**两个不同的实根、两个相等的实根、虚根等写出通解**，最终得出递推方程的解。

下面举一个非齐次线性递推关系两个不同实根的例子。

$$\begin{cases} f_n = 7f_{n-1} - 12f_{n-2} + 6n - 5 (n \geq 2) \\ f_0 = 5, f_1 = 13 \end{cases}$$

特征方程为：$X^2 - 7X + 12 = 0$
特征根为 $X_1 = 3\ X_2 = 4$ 为两个不等实根

对应齐次递推关系通解为$f_n^{'} = c_1 3^n + c_2 4^n$

因为是非齐次，于是设特解为$f_n^{''} = A + Bn$

非齐次递推关系通解可得 A = 2， B = 1 于是 $f_n = f_n^{'} + f_n^{''} = c_1 3^n + c_2 4^n + 2 + n$

再带入可得 $c_1 = 2$， $c_2 = 1$，所以通解是 $f_n = 2 \cdot 3^n + 4^n + 2 + n$

具体求法可参考：https://hanshuliang.blog.csdn.net/article/details/109267705

一、常系数线性齐次递推方程求解过程

1 . 特征方程 :

2 . 解特征根 :

3 . 构造递推方程的通解 :

4 . 求通解中的系数 :

三、常系数线性非齐次递推方程 特解形式 (n 的 t 次多项式 | 特征根不为 1)

1 . 特解形式 :

二、常系数线性齐次递推方程求解过程 (有重根下的通解形式)

1 . 特征根数 :

2 . 根据特征根写出通解中的项 :

(1) 组成 :

(2) 系数项 :

(3) 通解第 i 项 :

3 . 写出通解 :

(1) 通解项数 :

(2) 通解组成 :

(3) 最终结果 : $H(n) = \sum H_i(n)$

四、常系数线性非齐次递推方程 特解形式 (n 的 t 次多项式 | 特征根为 1)

五、常系数线性非齐次递推方程 特解形式 (非齐次部分是指数 | 底不为特征根)

六、常系数线性非齐次递推方程 特解形式 (非齐次部分是指数 | 底是特征根)

1. 求解递归关系：

（1）$a_n = 5a_{n-1} - 6a_{n-2} \quad (n>1, a_0=0, a_1=1)$

特征方程：$X^2 - 5X + 6 = 0$

$\Rightarrow X_1 = 2 \quad X_2 = 3$ 为不等实根

$a_n = C_1 \cdot 2^n + C_2 \cdot 3^n$

$\because a_0 = 0 \quad a_1 = 1$

齐次

$\Rightarrow \begin{cases} 0 = C_1 + C_2 \\ 1 = 2C_1 + 3C_2 \end{cases}$

$\Rightarrow \begin{cases} C_1 = -1 \\ C_2 = 1 \end{cases}$

$a_n = -2^n + 3^n$

$a_n =$

（2）$a_n = 5a_{n-1} - 6a_{n-2} + 2n \quad (n>1, a_0=0, a_1=1)$

特征方程 $X^2 - 5X + 6 = 0$

$X_1 = 2 \quad X_2 = 3$

$a_n' = C_1 \cdot 2^n + C_2 \cdot 3^n$

非齐次 $\Rightarrow a_n'' = An + B$（特解：从 $a_n = 5a_{n-1} - 6a_{n-2} + 2n$ 得）

$An + B = 5[A(n-1) + B] - 6[A(n-2) + B] + 2n$

$\Rightarrow A = 1 \quad B = \frac{7}{2}$

$a_n = C_1 \cdot 2^n + C_2 \cdot 3^n + n + \frac{7}{2}$

代入 $\Rightarrow \begin{cases} C_1 = -7 \\ C_2 = \frac{1}{2} \end{cases}$

$\Rightarrow a_n = -7 \cdot 2^n + \frac{1}{2} \cdot 3^n + n + \frac{7}{2}$

给出一个线性递推方程：

$$a_n + p_1 a_{n-1} + p_2 a_{n-2} + \cdots + p_k a_{n-k} = f(n)$$

若$f(n)$不为0，则式(8)称为非齐次线性递推方程。$f(n)$可以为常数，也可以为关于n的多项式，也可以为以n为指数的表达式。
对于这类方程，我们可以通过构造法，将它变成齐次线性递推方程。
构造t_n满足：

$$t_n + p_1 t_{n-1} + p_2 t_{n-2} + \cdots + p_k t_{n-k} = f(n)$$

其中t_n称为非线性递推方程的特解。

设$b_n = a_n - t_n$.
则b_n满足：

$$b_n + p_1 b_{n-1} + p_2 b_{n-2} + \cdots + p_k b_{n-k} = 0$$

b_n可以通过齐次线性递推方程求解。
它是齐次线性递推方程的通解。于是$a_n = b_n + t_n$，即非齐次线性递推方程的解可由齐次方程的通解和特解求得。

（3）$a_n = 5a_{n-1} - 6a_{n-2} + 3^n \quad (n>1, a_0=0, a_1=1)$

特征方程：（同上）

非齐次：$a_n'' = An \cdot 3^n$

$\Rightarrow An \cdot 3^n = 5A(n-1) \cdot 3^{n-1} - 6A(n-2) \cdot 3^{n-2} + 3^n$

$A = 3 \quad \Rightarrow a_n = C_1 \cdot 2^n + C_2 \cdot 3^n + n \cdot 3^{n+1}$

$\begin{cases} C_1 = \\ C_2 = \end{cases}$

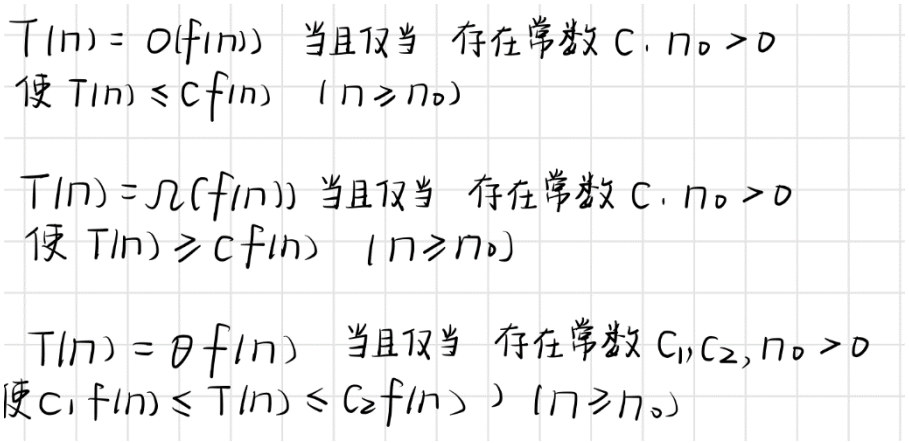
（略）

$\Rightarrow a_n =$

（4）$a_n = 5a_{n-1} - 6a_{n-2} + 3^n - 2^n \quad (n>1, a_0=0, a_1=1)$

非齐次：$a'' = An \cdot 3^n + Bn \cdot 2^n$

2.O()计算



第二章 递归与分治

1.递归的思考

不断缩小查找或者是计算范围，因为小问题更好解，再递推到大问题。
问题的关键在于递归公式的寻找，如何分解问题成为关键。

2.一些题目（节选）

(1) 汉诺塔 —— 递归

设三塔从左往右依次是 a，b，c，将 a 上的全部转移到 c 上(注意课本里是 a->b)
由数学归纳法，我们发现，每一步都能简化成为先把 n-1 个移到 b 上，把第 n 个移到 c 上
同样剩余 n-1 个由 b 移到 c 上
由此我们可以写出代码 (见 hanoi.c 文件)

(2)整数划分——递归

递归公式，按这个写就可以得到要求代码(见 Decomp.cpp)

$$f(n,m)=\begin{cases}1 & n=1,m=1\\ f(n,n) & n<m\\ 1+f(n,n-1) & n=m\\ f(n,m-1)+f(n-m,m) & n>m>1\end{cases}$$

3.分治的思考

有最优子结构性质的问题，可以考虑使用分治来解决。子问题相互独立用分治，如果不独立请使用动态规划。

4.分治的题目——经典算法(必须掌握)

(1)二分搜索

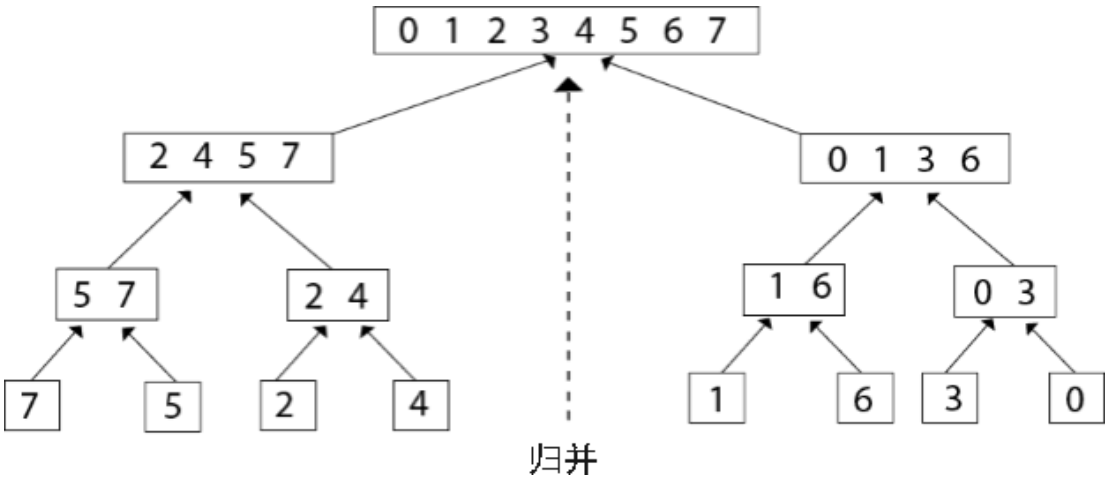
因为需要查找的数列是有序的(默认升序)，所以将 target 值与当前数组中间数值比较，不断**更新 left 与 right 的范围(查找范围)**就可以了
代码实现见 searchTwo.cpp 以及 searchTwo(非递归).cpp

(2)快速排序

快速排序是找出一个元素（理论上可以随便找一个）作为基准(pivot),然后对数组进行分区操作,**使基准左边元素的值都不大于基准值,基准右边的元素值 都不小于基准值**，如此作为基准的元素调整到排序后的正确位置。**递归快速排序**，将其他 n-1 个元素也调整到排序后的正确位置。最后每个元素都是在排序后的正确位置，排序完成。所以快速排序算法的核心算法是**分区操作**，即如何调整基准的位置以及调整返回基准的最终位置以便分治递归。
假设要排序的序列为
2 4 9 3 6 7 1 5 首先用 2 当作基准，使用 i j 两个指针分别从两边进行扫描，把比 2 小的元素和比 2 大的元素分开。首先比较 2 和 5，5 比 2 大，j 左移
2 2 4 9 3 6 7 1 5 比较 2 和 1，1 小于 2，所以把 1 放在 2 的位置
2 1 4 9 3 6 7 1 5 比较 2 和 4，4 大于 2，因此将 4 移动到后面
2 1 4 9 3 6 7 4 5 比较 2 和 7，2 和 6，2 和 3，2 和 9，全部大于 2，满足条件，因此不变
经过第一轮的快速排序，元素变为下面的样子
[1] 2 [4 9 3 6 7 5]
之后，在把 2 左边的元素进行快排，由于只有一个元素，因此快排结束。右边进行快排，递归进行，最终生成最后的结果。
代码实现见 quickSort.cpp

(3)归并排序

思路如下，拆分与二分搜索有相似之处，但是不会出现删除一半的现象，而是拆到最小时，开始一边合并一边回退。拆分相当简单，而合并我选择的做法是，开辟额外的空间对两段重排，最后放回原数组。



代码可参见：mergeSort.cpp

(4)线性时间排序

给定一个能够线性排序的集合（该集合中有 n 个元素）和 一个整数 k（1 ⩽ k ⩽ n） ，找出这 n 个元素中第 k 小的元素。，“线性时间选择”就是“选择问题”的“线性时间”算法
我的理解是：就是对快排的基准进行了设定，从而使快排的效率靠近其最高效率。
将 n 个元素，分成 n / 5 组，取出每组的中位数（第三小的元素）
取出 n / 5 个中位数的中位数（Select 函数可以取中位数）
利用快排中的分解函数 Partition()，以所求中位数为基准，划分 a[p : r] 为两段。
取其中一段进行递归。

5.其他应用

期中 2

设计一个算法，找出一个无序的数组中的最大两个数和最小的一个数的值。假定共有 n 个元素，使算法比较次数不超过[2n]。写出算法设计的思路，并分析算法的复杂度（可假定数组个数为 2^m),并给出算法 C 语言的伪代码实现）
我的思路是快排的想法，但好像记得 lxh 讲得是归并？

期中 3

平面最接近点对问题: 给定二维平面上乱序的 n (n>1)个坐标点，找出其中相距最近的两个坐标点(欧氏距离下)，并输出最近距离和任意一组最优解， 写出算法设计的思路，分析算法的复杂度，并给出算法 C 语言的伪代码实现

(1) 一维情况

先分为两段找最小，再找两端中间的两点的距离，看是否需要更新最小值。(即查看**(m-d，m]和[m，m+d)**)

(2) 二维情况

根据水平方向的坐标把平面上的 N 个点分成两部分 Left 和 Right。跟以往一样，我们希望这两个部分点数的个数差不多。假设分别求出了 Left 和 Right 两个部分中距离最近的点对之最短距离为 MinDist（Left）和 MinDist（Right），还有一种情况我们没有考虑，那就是点对中一个点来自于 Left 部分，另一个点来自于 Right 部分。最直接的想法，那就是穷举 Left 和 Right 两个部分之间的点对，这样的点对很多，最多可能有 $N * N/4$ 对。显然，穷举所有 Left 和 Right 之间的点对是不好的做法。是否可以只考虑有可能成为最近点对的候选点对呢？由于我们已经知道 Left 和 Right 两个部分中的最近点对距离分别为 MinDist（Left）和 MinDist（Right），如果 Left 和 Right 之间的点对距离超过 MDist = MinValue（MinDist（Left），MinDist（Right）），我们则对它们并不感兴趣，因为这些点对不可能是最近点对。

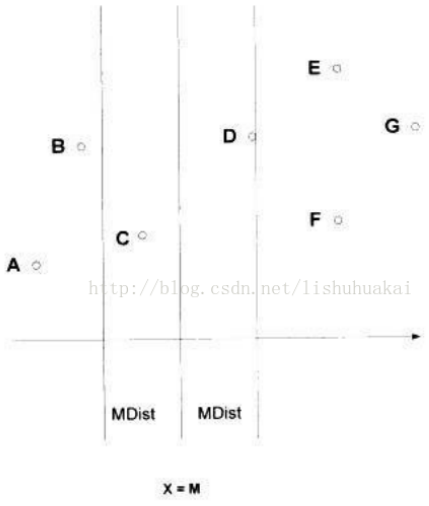


图 2-10　二维点分布示意图

如图 2-10 所示，通过直线 $x = M$ 将所有的点分成 $x < M$ 和 $x > M$ 两部分，在分别求出两部分的最近点对之后，只需要考虑点对 CD。因为其他点对 AD、BD、CE、CF、CG 等都不可能成为最近点对。也就是说，只要考虑从 $x = M - \text{Mdist}$ 到 $x = M + \text{MDist}$ 之

间这个带状区域内的最小点对，然后再跟 MDist 比较就可以了。在计算带状区域的最小点对时，可以按 Y 坐标，对带状区域内的顶点进行排序。如果一个点对的距离小于 MDist，那么它们一定在一个 MDist *（2 * Mdist）的区域内（如图 2-11 所示）：

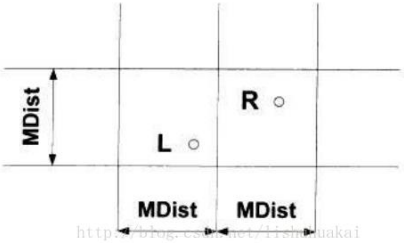


图 2-11 区域示意图

而在左右两个 Mdist * MDist 正方形区域内，最多都只能含有 4 个点。如果超过 4 个点，则这个正方形区域内至少存在一个点对的距离小于 Mdist，这跟 $x < M$ 和 $x > M$ 两个部分的最近点对距离分别是 MinDist（Left）和 MinDist（Right）矛盾。

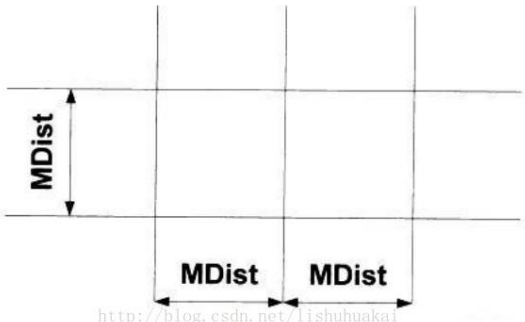


图 2-12 区域示意图

因此，一个 MDist *（2 * Mdist）的区域内最多有 8 个点（如图 2-12 所示）。对于任意一个带状区域内的顶点，只要考察它与按 Y 坐标排序且紧接着的 7 个点之间的距离就可以了。根据这个特点，我们可用 $O(N)$ 时间完成带状区域最近点对的查找。在这一步，需要注意的是：我们可以用归并排序法将带状区域的点按 Y 坐标排序。归并排序的过程与计算最近点对的算法结合在一起。

代码实现见程序 plaPoint.cpp

6.做一做

给定整数数组 nums 和整数 k，请返回数组中第 k 个最大的元素。
请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。
提醒：使用快排的思路去寻找
https://leetcode-cn.com/problems/kth-largest-element-in-an-array/

第三章 动态规划

1.什么是动态规划

通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。动态规划常常适用于有**重叠子问题和最优子结构性质**的问题。
最核心的思想，就在于**拆分子问题，记住过往，减少重复计算**。
动态规划有几个典型特征，**最优子结构、状态转移方程、边界、重叠子问题**。

2. 动态规划的两种思路

带备忘录的递归解法（自顶向下）

斐波那契数列

暴力递归？ 不断计算重复的值，太浪费递归所用空间
解决手段： 对重复的值用数组加以存储

```
class Solution {
public:
    int fib(int n) {
        vector<int> dp;
        dp.push_back(0);
```

```
        dp.push_back(1);
        for(int i = 2; i <= n; i++)
        {
            dp.push_back( dp[i-1]+ dp[i-2]);
        }
        return dp[n];
    }
};
```

自底向上的动态规划

动态规划跟带备忘录的递归解法基本思想是一致的，都是减少重复计算，时间复杂度也都是差不多。带备忘录的递归，是从 f**(10)往 f(1）方向延伸求解的，所以也称为自顶向下的解法**。动态规划从较小问题的解，由交叠性质，逐步决策出较大问题的解，**它是从 f(1)往 f(10）**方向，往上推求解，所以称为自底向上的解法。

青蛙跳台阶问题

https://leetcode-cn.com/problems/climbing-stairs/solution/70-pa-lou-ti-by-huyii-5lb6/

```
class Solution {
public:
    //动态规划方程为 dp[n] = dp[n-1] + dp[n-2];
    int climbStairs(int n) {
        if(n == 1)
            return 1;
        if(n == 2)
            return 2;
        int dp1 = 1;
        int dp2 = 2;
        int result = 0;
        for(int i = 2; i < n; i ++)
        {
            result = dp2 + dp1;
            dp1 = dp2;
            dp2 = result;
        }
        return result;
    }
};
```

3.动态规划的经典题目

(1)最长公共子序列

一个字符串的**子序列**是指这样一个新的字符串：它是由原字符串在**不改变字符的相对顺序的情况下删除某些字符**（也可以不删除任何字符）后组成的新字符串。(区分于子集！)
Leetcode 链接：https://leetcode-cn.com/problems/longest-common-subsequence/

动态规划方程含义

dp[i][j] 表示S_1与S_2最长子序列长度为 dp[i][j]　　$S_1 = A_{0-i}$ $S_2 = B_{0-j}$
***本题中字符串从 1 开始到 len**

转移方程

When A[i] = B[j]
dp[i][j] = dp[i-1][j-1] + 1　如果新增结尾的两个字符相等，相同序列长度直接加 1
eg: A：acdf　dp[2][0] = 1;　dp[3][1] = dp[2][0] + 1
　　B：df
When A[i] != B[j]
dp[i][j] = max（dp[i][j-1]，dp[i-1][j]）　如不等，那么就在已求过的 dp[i][j-1]，dp[i-1][j]选更大值(求最长的，所以肯定在次长里选)

边界条件和初始化考虑

i = 0 时 表示的是S_1中取空字符串跟S_2最长公共子序列，结果肯定为 0.
j = 0 同理

遍历方向与范围

由转移方程中不等的情况可以看出，i, j 均为从小到大遍历，而 i = 0 与 j = 0 被初始化，所以都从 1 开始就好

代码

```
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
       int len1 = text1.size(), len2 = text2.size();
```

```
        vector<vector<int>> dp(len1+1, vector<int>(len2+1, 0));
        for(int i = 1; i <= len1; i++)
        {
            for(int j = 1; j <= len2; j++)
            {
                if(text1[i-1] == text2[j-1])
                    dp[i][j] = dp[i-1][j-1] + 1;
                else
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }

        }
        return dp[len1][len2];
    }
};
```

(2)最长递增子序列

给你一个整数数组 nums ， 找到其中最长严格递增子序列的长度。
子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。
LeetCode 链接：https://leetcode-cn.com/problems/longest-increasing-subsequence

动态规划方程含义

Dp[i] 表示n_0 − n_i中最长递增子序列长度

转移方程

If nums[i] > nums[j] && j < i
dp[i] = max(dp[i], dp[j] + 1) 即从以 0 开始，长度为 0-(i-1)的序列进行遍历，看是否能连接上 i

边界条件和初始化考虑

所有全部为 0， 方便 max 计算

遍历方向与范围

i 0-n 由小到大
j 0-i 由小到大

代码

```
class Solution {
 public:
    int lengthOfLIS(vector<int>& nums) {
        int n = (int)nums.size();
        if (n == 0) {
            return 0;
        }
        vector<int> dp(n, 0);
        for (int i = 0; i < n; ++i) {
            dp[i] = 1;
            for (int j = 0; j < i; ++j) {
                if (nums[j] < nums[i]) {
                    dp[i] = max(dp[i], dp[j] + 1);
                }
            }
        }
        return *max_element(dp.begin(), dp.end());
    }
};
```

(3)最长公共子串

最长公共子串（Longest Common Substring）与最长公共子序列（Longest Common Subsequence）的区别：子串要求在原字符串中是连续的，而子序列则只需保持相对顺序一致，并不要求连续。例如 X = {a, Q, 1, 1}; Y = {a, 1, 1, d, f}那么，{a, 1, 1}是 X 和 Y 的最长公共子序列，但不是它们的最长公共字串。

动态规划方程含义

dp[i][j] 表示以 i 结尾的S_1与以 j 结尾的S_2最长子集长度为 dp[i][j] $S_1 = A_{0-i}$ $S_2 = B_{0-j}$
注意!!! **最长子串一定是从结尾往前推的**
Eg: A：sdcdh

B：dcd　　Dp[3][2] = 3　　dp[3][1] = 0

***本题中字符串从 1 开始到 len**

转移方程

When A[i] = B[j]
dp[i][j] = dp[i-1][j-1] + 1
When A[i] ！= B[j]
dp[i][j] = 0

边界条件和初始化考虑

初始化直接全部为 0，　可以省去判断时的赋值功能

遍历方向与范围

遍历方向 i，j 均是由小到大

代码

见 getLongest.cpp

(4)0-1 背包

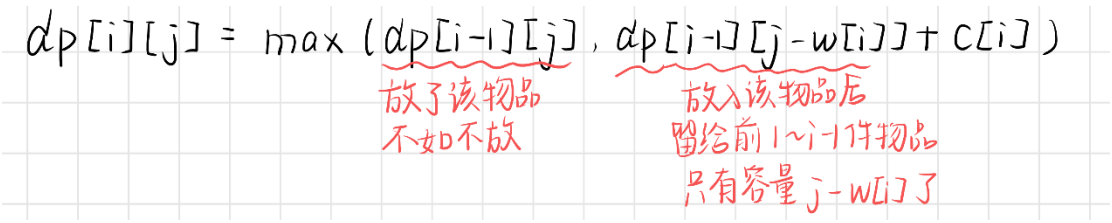
有 n 件物品和一个容量为 N 的背包。第 i 件物品的费用是 c[i]，价值是 w[i]。每件物品只能放一次，求解将哪些物品装入背包可使价值总和最大。

动态规划方程含义

dp[i][j] i 代表放到了第几件物品(无论前面放没放进去), j 则代表背包的剩余容量

转移方程

f (j >= w[i])//剩余空间足够
　　dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + c[i]);// 决 定 是 否 放 入




```
if (j < w[i]
    dp[i][j] = dp[i - 1][j];//肯定放不下
```

边界条件和初始化考虑

I 为 0 的情况下，即代表一件东西也没有放，所以 dp[0][j]全初始化为 0 就可以了
J 为 0 则表示背包没有空闲了，所以 dp[i][0]也全为 0

遍历方向与范围

外层的从 1—i，代表从第一件物品放到最后一件物品
内层的则由 N-0 代表背包由满到空
最大值一定存放在 dp[n][N]

代码

见 0-1bag.cpp(其中包括一维的简化)

(5)完全背包

n 种物品和一个体积为 N 的背包，每种物品都有无限件可用。第 i 件物品的体积是 w[i]，价值是 c [i]。求解将哪些物品装入背包可使这些物品的体积总和不超过背包体积，且价值总和最大。

思路

动态规划：是 0-1 背包一维解法的简化，具体解释见代码
贪心：后面再说。。。

代码

见 completeBag.cpp

(6)多重背包（了解一下，这个感觉用的比较少）

有 N 种物品和一个体积为 V 的背包。**第 i 种物品最多有 n[i]件可用**，每件体积是 volume[i]，价值是 value[i]。求解将哪些物品装入背包可使这些物品的体积总和不超过背包体积，且价值总和最大。

思路

动态规划：是 0-1 背包的一维解法加了数量限制

代码

见 multiBag.cpp

4.其他应用（ppt 中与提示内容)

矩阵连乘

给你一系列的矩阵 A1,A2,A3,……,An 和一系列的整数 P0,P1,P2……,Pn，每个矩阵 Ai 的规模为 Pi-1 * Pi。现在，请你计算这些矩阵连乘所需要的最少的计算次数是多少?
M[I,j] 的值为计算 Ai...j 所需要要的最少的计算次数，因此，这个最少计算次数的问题可以用下面的递归式来描述

$$m[i,j]=\begin{cases}0 & i=j,\\ \min_{i\le k<j}\left(m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\right) & i<j\end{cases}$$

凸多边形最优三角剖分

给定凸多边形 P，以及定义在由多边形的边和弦组成的三角形上的权函数 w。要求确定该凸多边形的三角剖分，使得该三角剖分中诸三角形上权之和为最小。

设 **t[i][j],1<=i<j<=n 为凸多边形{Vi-1,Vi……Vj}的最优三角剖分所对应的权值函数值，即其最优值**。最优剖分包含三角形 Vi-1VkVj 的权，子多边形{Vi-1,Vi……Vk}的权，子多边形{Vk,Vk+1……Vj}的权之和。

$$t[i][j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{t[i][k]+t[k+1][j]+w(v_{i-1}v_kv_j)\} & i<j\end{cases}$$

期中 4

给定两个由核苷酸构成的 DNA 序列 x 和 y,对齐的方式是**将空格分别插入到 x 和 y 序列中，得到具有相同长度的对齐后的序列 xe 和 ye;**

空格可以插入到任意的位置(包括两端),但是相同位置不能同时为空格,即不存在 xe|i|和 ye[i]同时为空格的情况。再为对齐后的序列的每个位置打分，总分为每个位置得分之和。

具体的打分规则如下:;

a、如果 xe[i]= ye[i]同且均不为空格，得 3 分:

b、如果 xe[i] != ye[i]同且均不为空格，得 1 分:

c、如果 xe[i] 或 ye[i]是空格，得 0 分:

求给定原序列 x 和 y 的一个对齐方案，使得该对齐方案的总分最高

(1)给出**计算最佳对齐方案总分值的算法**，并分析算法复杂度，写出类 C 的伪代码:

(2)给出其中一个序列最佳对齐方案的**生成方式** (仅计算公式) .

(1) 分析题目，我们可以发现得分主要来源于相同的两核苷酸，本题的实质与求一个最长公共子序列十分相似。于是我们按照这个思路继续往下分析。

价值 C[i][j]的动态规划方程可以写为

$$C[i][j]=\begin{cases}0 & i=0\ //\ j=0\\ c[i-1][j-1]+3 & X[i]=y[i]\\ \max\{c[i-1][j],c[i][j-1],c[i-1][j-1]+1\} & X[i]!=y[j]\end{cases}$$

(2)那么怎么去补全空格能得到最高的得分?

1、情况 a 用“=”字符标记;

2、情况 b 用“~”字符标记;

3、情况 c 用“*”字符标记,但是情况 c 实际上可以细分为两种情况: xe[i]为空格时用“+”标记,ye[i]为空格时用“-”号标记。这样用“+”和“-”细分的表示相比于统一用“*”来表示，本质的区别在于让对齐方案具有所谓的“方向性”，后面会看到这样的细分对于算法的实现有一定的好处。

用 R(i,j)表示序列 X[0]...X[i]和序列 Y[0].Y[j]的最优对齐方案的对齐规则字符串

X-A　Y-B

$$R(i,j)=\begin{cases}\overbrace{"+\ldots+"}^{j}, 如果i=0,\\ \overbrace{"-\ldots-"}^{i}, 如果j=0,\\ R(i-1,j-1)+"=", 如果A[i]=B[j]\neq 空格\\ R(i-1,j-1)+"\sim", \text{ 如果空格}\neq A[i]\neq B[j]\neq 空格\\ R(i-1,j)+"-", \text{ 如果}B[j]=空格\\ R(i,\ j-1)+"+", \text{ 如果}A[i]=空格\end{cases}$$

涉及了一些最短编辑距离问题：https://www.cnblogs.com/qcblog/p/7820139.html

6.做一做

(1)最长回文子串

题目与链接

链接：https://leetcode-cn.com/problems/longest-palindromic-substring/solution/
给你一个字符串 s，找到 s 中最长的回文子串

思路

含义：Dp[i][j] 表示$S_i \sim S_j$这一段子串是否为回文子串，如果不是则为 false，是则设为 true
转移方程：dp[i][j] = dp[i+1][j-1] && true; s[i] = s[j]
dp[i][j] = false; s[i] != s[j]
边界条件(初始化)：如果出现了 i > j，由于实际上的情况只能是相邻两位，所以直接设为 true

(2)最大子段和

题目与链接

https://leetcode-cn.com/problems/maximum-subarray/
给你一个整数数组 nums ，请你找出一个具有最大和的**连续**子数组（子数组最少包含一个元素），返回其最大和。

思路

含义：Dp[i]表示以 i 结尾的具有最大和的连续子数组
转移方程：dp[i] = max(dp[i-1] +nums[i], nums[i]);

(3)最大正方形

题目与链接

https://leetcode-cn.com/problems/maximal-square/
在一个由 '0' 和 '1' 组成的二维矩阵内，找到只包含 '1' 的最大正方形，并返回其面积。

思路

含义：dp[i][j] 以 i,j 为右下角的最大正方形
转移方程：dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1 //如果 matrix[i][j] == 1
dp[i][j] = 0 //如果 matrix[i][j] == 0

(4)打家劫舍

题目与链接

https://leetcode-cn.com/problems/house-robber/
你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。
给定一个代表每个房屋存放金额的非负整数数组，计算你不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额

思路

含义：dp[i] 偷到第 i 个房子的最高金额
转移方程：dp[i] = max(dp[i-1], dp[i-2] + nums[i]);

(5)最长有效括号

题目与链接

https://leetcode-cn.com/problems/longest-valid-parentheses/
给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

思路

含义：dp[i] 表示 i 结尾的字符串中最长有效括号子串长度
转移方程：dp[i] = 0 s[i] = '('
dp[i] = 2 + dp[i-2] s[j] = ')' &&s[j-1] = '('
dp[i] = dp[i-1] + dp[i-dp[i-1] - 2] + 2 s[j] = ')' &&s[j-1] = '）' &&s[j-dp[i-1]-1] = '('
//注意边界条件，上述式子中未考虑越界

(6)零钱兑换——背包问题

https://leetcode-cn.com/problems/coin-change/

(7)单词拆分——完全背包问题

https://leetcode-cn.com/problems/word-break/

第四章 贪心

1.选择贪心法

贪心法顾名思义，即是在当前情况下做出看起来最好的选择，并不从整体来考虑。
所以简单来说，选择贪心法是**因为局部最优解能推出全局最优解**，每次的选择可依赖以前作出的选择，但不依赖后续选择。
既然动态规划与贪心法都要求题目有最优子结构，在选择算法时，还需要考虑什么呢？
动态规划算法中，每步所做的选择往往**依赖于相关子问题的解**，因而只有在解出相关子问题时才能做出选择。而**贪心算法**，仅在**当前状态下做出最好选择**，即局部最优选择，然后再去解做出这个选择后产生的相应的子问题。

2.经典问题

(1)开会问题

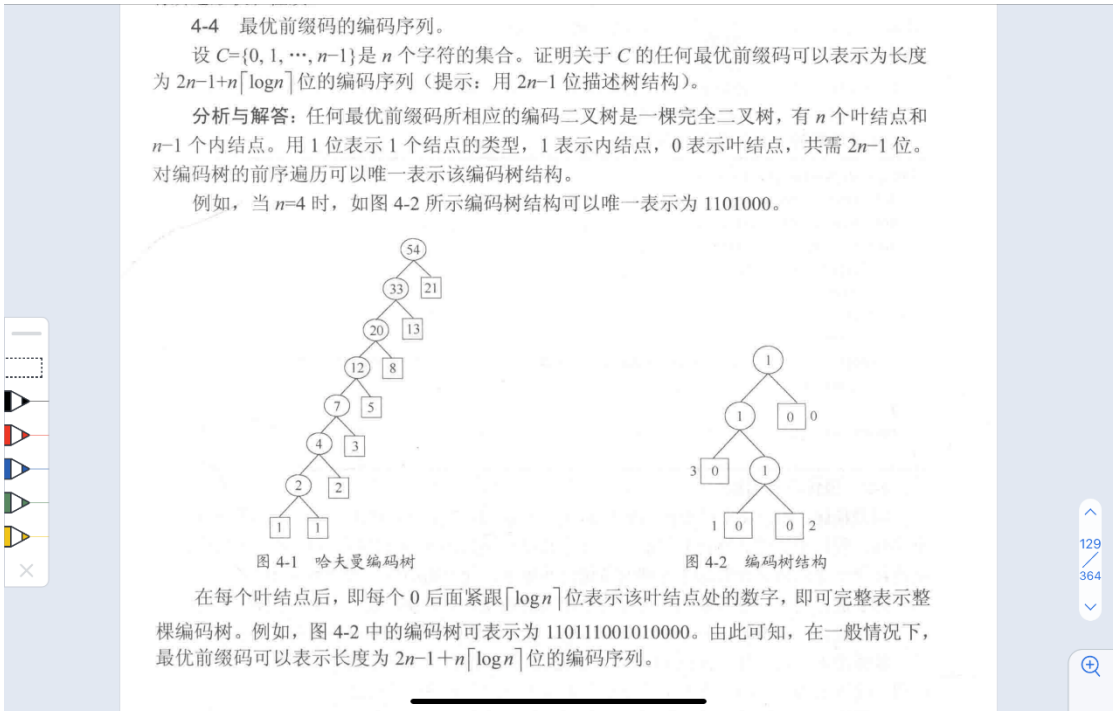
设有 N 个活动时间集合，每个活动都要使用同一个资源，比如说会议场，而且同一时间内只能有一个活动使用，**每个活动都有一个使用活动的开始 si 和结束时间 fi**，即他的使用区间为（si,fi）,现在要求你分配活动占用时间表，即哪些活动占用该会议室，哪些不占用，使得他们不冲突，**要求是尽可能多的使参加的活动最大化**
按照从小到大排序，**挑选出结束时间尽量早的活动，并且满足后一个活动的起始时间晚于前一个活动的结束时间**，全部找出这些活动就是最大的相容活动子集合。
代码详见：actiSchudule.cpp

(2)哈夫曼编码（作业中的证明）

表示最优前缀码的二叉树总是一棵完美二叉树，即树中的每一个结点都有两个儿子结点。
平均码长定义为：$B(T) = \sum_{c \in C} f(c) d_T(c)$

HuffmanTree 的形成：每次选两个最小频率为子树。

由于用最小堆实现了优先队列 Q，初始化 Q 用时 O(logn), n-1 次合并用时 O(nlogn)



4-4　最优前缀码的编码序列。

设 $C=\{0, 1, \cdots, n-1\}$ 是 n 个字符的集合。证明关于 C 的任何最优前缀码可以表示为长度为 $2n-1+n\lceil \log n \rceil$ 位的编码序列（提示：用 $2n-1$ 位描述树结构）。

分析与解答：任何最优前缀码所相应的编码二叉树是一棵完全二叉树，有 n 个叶结点和 $n-1$ 个内结点。用 1 位表示 1 个结点的类型，1 表示内结点，0 表示叶结点，共需 $2n-1$ 位。对编码树的前序遍历可以唯一表示该编码树结构。

例如，当 $n=4$ 时，如图 4-2 所示编码树结构可以唯一表示为 1101000。

图 4-1　哈夫曼编码树

图 4-2　编码树结构

在每个叶结点后，即每个 0 后面紧跟 $\lceil \log n \rceil$ 位表示该叶结点处的数字，即可完整表示整棵编码树。例如，图 4-2 中的编码树可表示为 110111001010000。由此可知，在一般情况下，最优前缀码可以表示长度为 $2n-1+n\lceil \log n \rceil$ 位的编码序列。

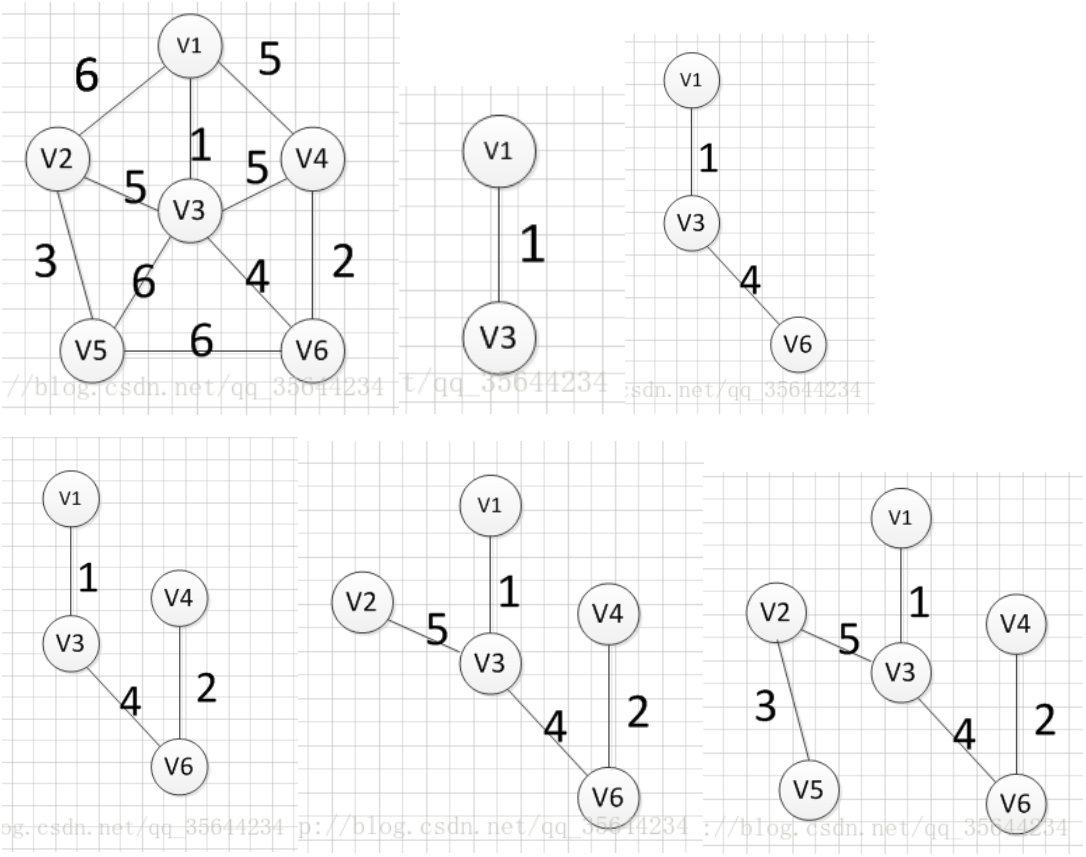
(3)迪杰斯特拉算法

太熟悉，所以不说了~

(4)最小生成树

a. prim 算法

首先就是从图中的**一个起点 a 开始，把 a 加入 U 集合，然后，寻找从与 a 有关联的边中，权重最小的那条边并且该边的终点 b 在顶点集合**：（V-U）中，我们也把 b 加入到集合 U 中，并且输出边（a，b）的信息，这样我们的集合 U 就有：{a,b}，然后，我们寻找与 a 关联和 b 关联的边中，权重最小的那条边并且该边的终点在集合：（V-U）中，我们把 c 加入到集合 U 中，并且输出对应的那条边的信息，这样我们的集合 U 就有：{a,b,c}这三个元素了，一次类推，直到所有顶点都加入到了集合 U。



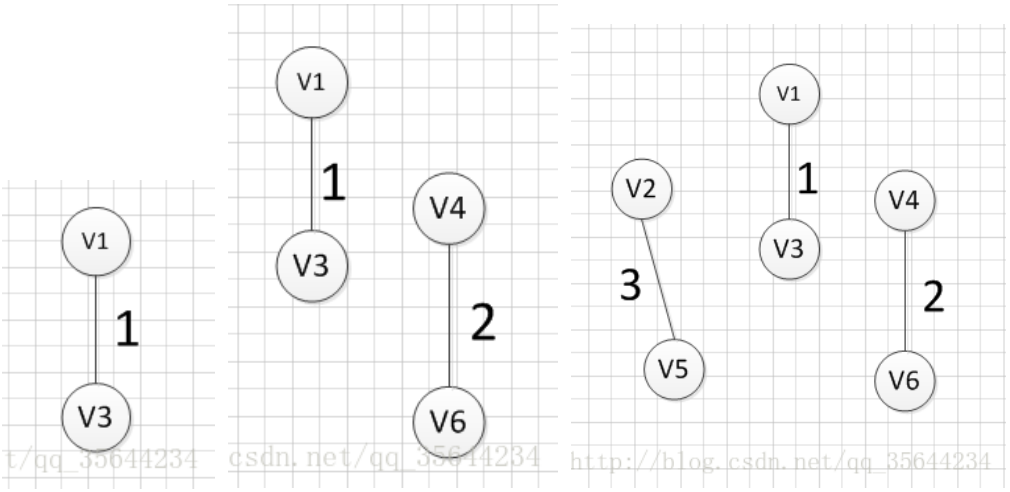
b. Kruskal 算法

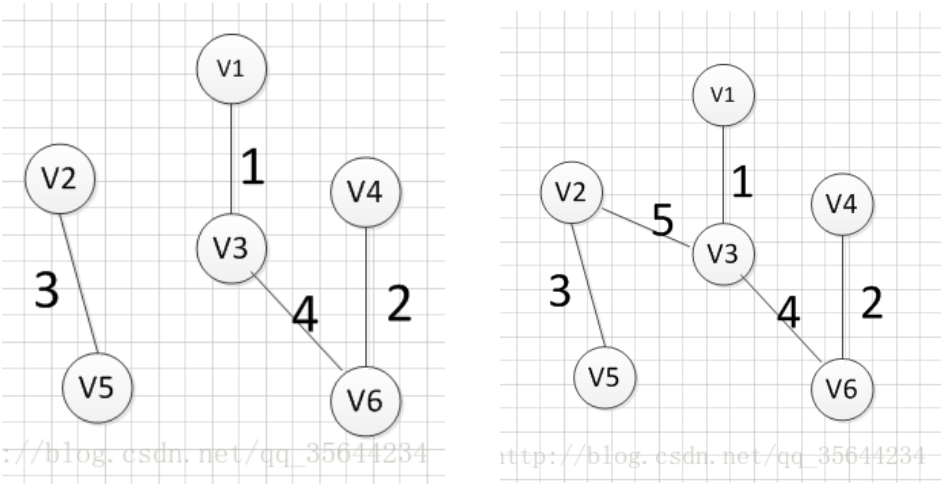
算法思路:

（1）将图中的所有边都去掉。

（2）**将边按权值从小到大的顺序添加到图中**，保证添加的过程中**不会形成环**

（3）重复上一步直到连接所有顶点，此时就生成了最小生成树。这是一种贪心策略。





(5)线段覆盖

在一维空间中告诉你 N 条线段的起始坐标与终止坐标，要求求出这些线段一共覆盖了多大的长度。
代码见 lineCovers.cpp

(6)多机调度问题(作业)

设有 n 个独立的作业{1, 2, …, n}，由 m 台相同的机器{M1, M2, …, Mm}进行加工处理，作业 i 所需的处理时间为 ti（1≤i≤n），每个作业均可在任何一台机器上加工处理，但不可间断、拆分。多机调度问题要求给出一种作业调度方案，使所给的 n 个作业在尽可能短的时间内由 m 台机器加工处理完成。
贪心法求解多机调度问题的贪心策略是**最长处理时间作业优先，即把处理时间最长的作业分配给最先空闲的机器**，这样可以保证处理时间长的作业优先处理，从而在整体上获得尽可能短的处理时间。
代码见 multSchedule.cpp

(5)最优装载

有一批集装箱要装上一艘载重量为 c 的轮船。其中集装箱 i 的重量为 Wi，最优装载问题要求确定在装载体积不受限制的情况下，将尽可能多的集装箱装上轮船。
每次都选最小重量的上去就可以！
考查重点放在了 sort 上

3.做一做

(1)跳跃游戏

题目与链接

https://leetcode-cn.com/problems/jump-game/
给定一个非负整数数组 nums ，你最初位于数组的第一个下标 。
数组中的每个元素代表你在该位置可以跳跃的最大长度。
判断你是否能够到达最后一个下标

思路

依次遍历数组中的每一个位置，并**实时维护最远可以到达**的位置

(2)跳跃游戏 II

题目与链接

https://leetcode-cn.com/problems/jump-game-ii/
给你一个非负整数数组 nums ，你最初位于数组的第一个位置。
数组中的每个元素代表你在该位置可以跳跃的最大长度。
你的目标是使用最少的跳跃次数到达数组的最后一个位置。
假设你总是可以到达数组的最后一个位置。

思路

在能跳跃到的范围中贪心地去寻找每次能跳到的最远的点，以减少跳跃次数

(3)加油站

题目与链接

https://leetcode-cn.com/problems/gas-station/
在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i] 升。
你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。你从其中的一个加油站出发，开始时油箱为空。
如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1

思路

如果 i 站停下，即剩余油不够汽车从 i 开往 i+1，那么从起点到 i 之间的任何一个节点都无法到达 i+1 结点。减少了更新 i 的次数

第五章 回溯法

1.什么是回溯法

回溯法最好懂的理解就是走迷宫，不断地试错和回退是回溯法的核心思想。而在实际实现中，考虑到算法的效率，我们会对回溯条件进行一些限制，以减少不必要的回溯。

2.回溯法能套用的模板

(1)子集树

当所给的问题是从 **n 个元素的集合 S** 中找出满**足某种性质的子集**时，相应的解空间树称为子集树。子集树通常有 2^n 个叶结点(完全二叉树)

```
void backtrack(int i)
{
    if(i>t)
    {
        //找到了一个解,记录一下
        return ;
    }
    if(Constraint1() && Bound1())//左子树剪枝
    {
        x[i]=1;
```

```
            //考虑搜索左子树
            backtrack(i+1);
            //回退了!维护好
        }
        if(Constraint0() && Bound0())//右子树剪枝
        {
            x[i]=0;
            //考虑搜索右子树
            backtrack(i+1);
        }
return ;
}
```

(2)排列树

当所给的问题是确定 n 个元素满足**某种性质的排列**时，相应的解空间树称为排列树。排列树通常有 n!个叶节点。也就是说 **n 个元素的每一个排列都是解空间中的一个元素**。

```
Void backtrack()
{
    if(回溯条件)
    {
        需要做的事情;
        return;
    }

    for()
    {
        执行操作;
        Backtrack();
        撤销操作;
    }
}
```

3.经典问题

(1)DFS 深度优先搜索

对所有结点进行访问，访问过的结点进行标记

(2)N 皇后

参见博客：https://blog.csdn.net/zhang245754954/article/details/52612784

(3)全排列

一串数各不相同，输出所有它的排列方式

(4)装载问题-子集树

有一批共 n 个集装箱要装上 2 艘载重量分别为C_1和C_2的轮船，其中集装箱 i 的重量为W_i，且$\sum_{i=1}^{n} W_i \leq C_1 + C_2$
装载问题要求确定**是否有一个合理的装载方案可将这个集装箱装上**这 2 艘轮船。如果有，找出一种装载方案。
类 0-1 背包问题， 把其中一个船拿出当包即可
见代码 getLoad.cpp(主要看 seeHere 部分)

(5)批处理作业调度-排列树

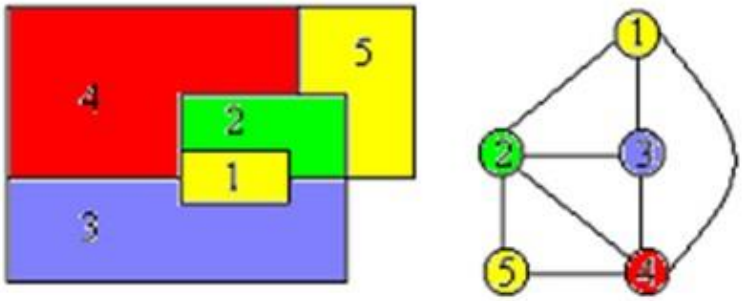
给定 n 个作业的集合{J1,J2,…,Jn}。每个作业**必须先由机器 1 处理，然后由机器 2 处理。**作业 Ji 需要机器 j 的处理时间为 tji。对于一个确定的作业调度，设 Fji 是作业 i 在机器 j 上完成处理的时间。所有作业在机器 2 上完成处理的时间和称为该作业调度的完成时间和。批处理作业调度问题要求对于给定的 n 个作业，**制定最佳作业调度方案，使其完成时间和达到最小。时间应该怎么去求**?
作业 i 在机器 1 上完成的时间是连续的，所以是直接累加就可以。但对于机器 2 就会产生两种情况，第一种，在求作业 2 在机器 2 上完成的时间时，由于作业 2 在机器 1 上还没有完成，这就需要先等待机器 1 处理完；第二种，在求作业 2 在机器 2 上完成的时间时，作业 2 在机器 1 早已完成，无需等待，直接在作业 1 被机器 1 处理之后就能接着被处理。
综上，我们可以得到如下表达式

```
if (F2[i-1] > F1[i])
    F2[i] = F2[i-1] + t[2][i]
else
    F2[i] = F1[i] + t[2][i]
```

具体代码见 batSolu.cpp

(6)图的 m 着色问题

给定无向连通图 G 和 m 种不同的颜色。用这些颜色为图 G 的各顶点着色，每个顶点着一种颜色。是否有一种着色法使 G 中每条边的 2 个顶点着不同颜色。这个问题是图的 m 可着色判定问题。若一个图最少需要 m 种颜色才能使图中每条边连接的 2 个顶点着不同颜色，则称这个数 m 为该图的色数。求一个图的色数 m 的问题称为图的 m 可着色优化问题。



回溯条件：如果当前**节点所有颜色都不可行**，结束对子树的遍历，返回

(7)TSP 旅行商问题

一个商品推销员要去若干个城市推销商品，该推销员从一个城市出发，需要经过所有城市后，回到出发地。应如何选择行进路线，以使总的行程最短。
两个剪枝条件:
1、利用约束函数减去不满足约束条件的子树。
2、利用限界函数减去得不到最优解的子树。
约束为**当前节点到下一节点的长度不为 0**,限界为走过的长度+当前要走的长度之和小于最优长度

4.做一做

(1) 括号生成-排列树

题目与链接

https://leetcode-cn.com/problems/generate-parentheses/
数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且有效的 括号组合。

思路

先对 open<n 进行回溯，在对 close<open 进行回溯，当总长度正确，则存入输出

(2)全排列 II-排列树

题目与链接

https://leetcode-cn.com/problems/permutations-ii/
给定一个可包含重复数字的序列 nums ， 按任意顺序 返回所有不重复的全排列。

思路

我们选择对原数组排序，保证相同的数字都相邻，然后每次填入的数一定是这个数所在重复数集合中「从左往右第一个未被填过的数字」

(3)子集-子集树

题目与链接

https://leetcode-cn.com/problems/subsets/
给你一个整数数组 nums ，数组中的元素 互不相同 。返回该数组所有可能的子集（幂集）。
解集 不能 包含重复的子集。你可以按 任意顺序 返回解集。

思路

代码框架

```
vector<int> t;
void dfs(int cur, int n) {
    if (cur == n) {
        // 记录答案
        // ...
        return;
    }
    // 考虑选择当前位置
    t.push_back(cur);
    dfs(cur + 1, n, k);
    t.pop_back();
    // 考虑不选择当前位置
    dfs(cur + 1, n, k);
}
```

(4)分割回文串

题目与链接

https://leetcode-cn.com/problems/palindrome-partitioning/
给你一个字符串 s，请你将 s 分割成一些子串，使每个子串都是 回文串 。返回 s 所有可能的分割方案。
回文串 是正着读和反着读都一样的字符串。

思路

回溯+动态规划