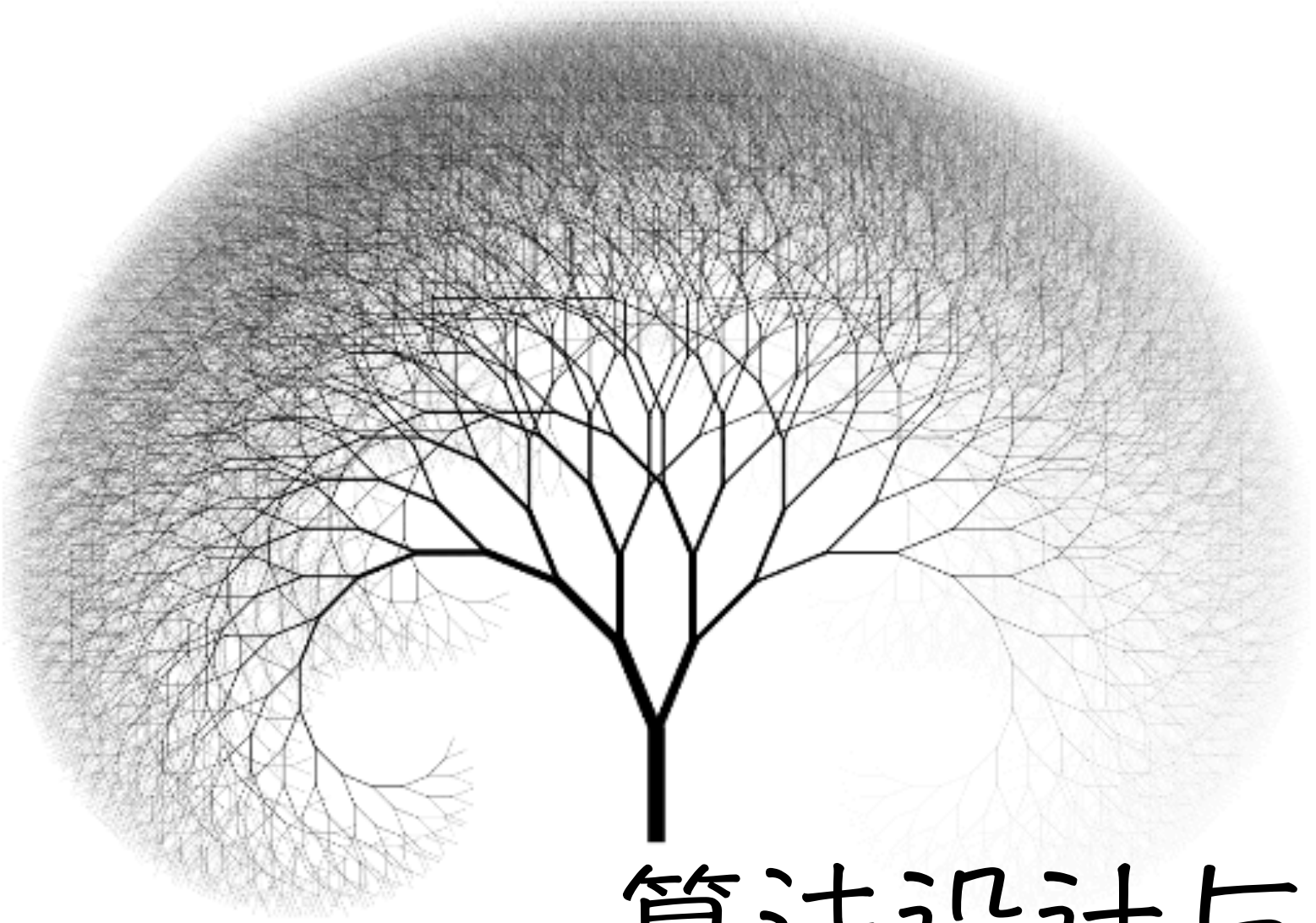


算法设计与分析

刘 安
苏州大学 计算机科学与技术学院
http://web.suda.edu.cn/anliu/

蛮力（穷举）搜索

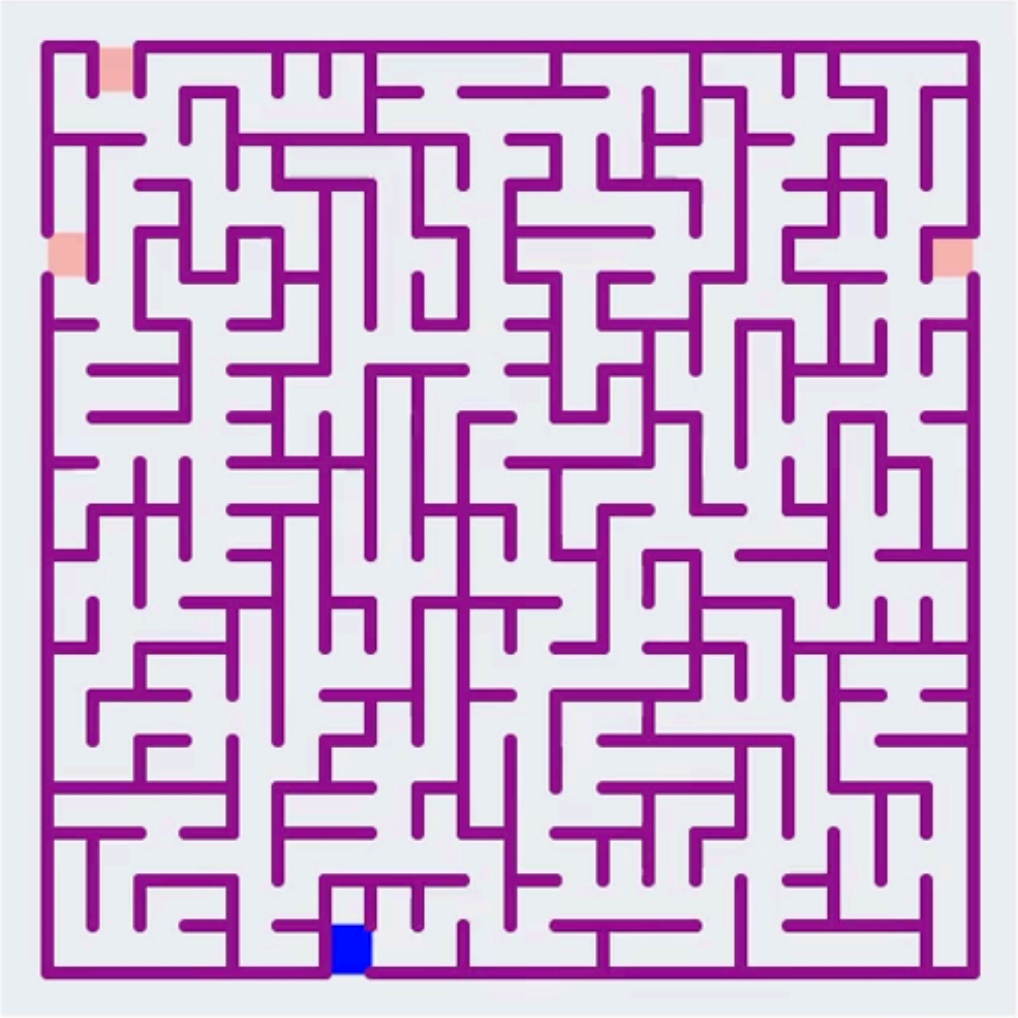
- 蛮力（穷举）搜索：brute-force (exhaustive) search
  - 系统地枚举所有可能的解并检查其合法性
  - 所有可能的解构成了问题的解空间
  - 独立集问题
    - 规模为k：穷举n个顶点所有规模为k的子集，$O(n^k)$
    - 最大独立集：穷举n个顶点所有的子集，$O(n^2 \cdot 2^n)$
  - 旅行商问题：穷举n个顶点所有的排列，$O(n \cdot n!)$
- 目前主流CPU在1秒内可以进行100万次搜索
  - 20个元素的子集数
  - 10个元素的排列数

回溯 backtracking

- 一种在解空间中系统搜索可行解的方法
- 问题的解：一个长度最多为n的决策序列$A[1..n]$
- 如果$A[1..i-1]$已经是一个解，结束当前递归调用
- 否则，确定第i步可用的候选方案C_i
  - 注意：C_i的值可能取决于过去的决策，即$A[1..i-1]$
  - 如果$C_i = \varnothing$，结束当前递归调用
    - 避免无效搜索，也称为剪枝（pruning）
  - 否则依次尝试方案$c \in C_i$
    - 第i步采用的方案确定之后（即$A[i] = c$），原问题就归约为一个新的子问题（确定长度最多为$n-i$的决策序列），递归求解

回溯

- 构造所有子集
- 构造所有排列
- 构造所有路径
- n皇后
- 子集和
- 文本分割
- 最长递增子序列
- 最优二叉搜索树



构造所有子集

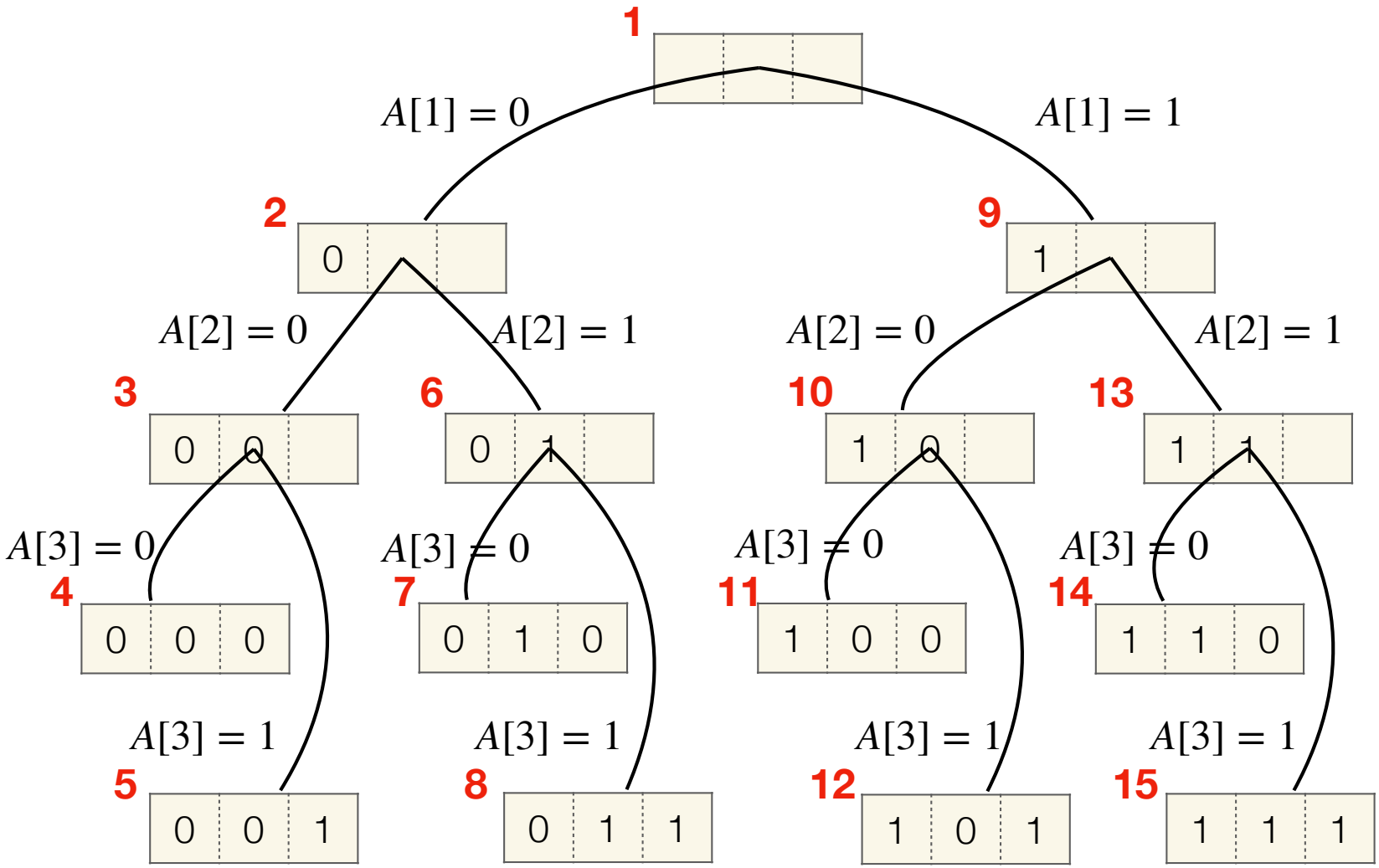
- 如何构造集合$\{1,2,\cdots,n\}$的所有子集
- 构造一个子集可以看成是一个长度为n的决策序列
  - 第i步决定是否将整数i放入目标子集中
    - 每一步只有两种候选方案：放、不放
    - 每一步的候选方案与之前的决策没有关系
  - 每个长度为n的决策序列对应一个子集
  - 考虑$n = 3$
    - 决策序列：放，不放，放 $\Rightarrow \{1, 3\}$
    - 决策序列：不放，不放，不放 $\Rightarrow \varnothing$

算法实现

- 全局数组$A[1..n]$存放所有的决策
  - $A[i]=0$，不把元素i放入子集
  - $A[i]=1$，把元素i放入子集
- $i>n$时，所有n个元素是否放入子集都已确定，找到一个解

```
def generate_subset(i):
    if i > n:
        print(A[1:])
    else:
        for c in range(2):
            A[i] = c
            generate_subset(i + 1)

n = 3
A = [None] * (n + 1)
generate_subset(1)
```

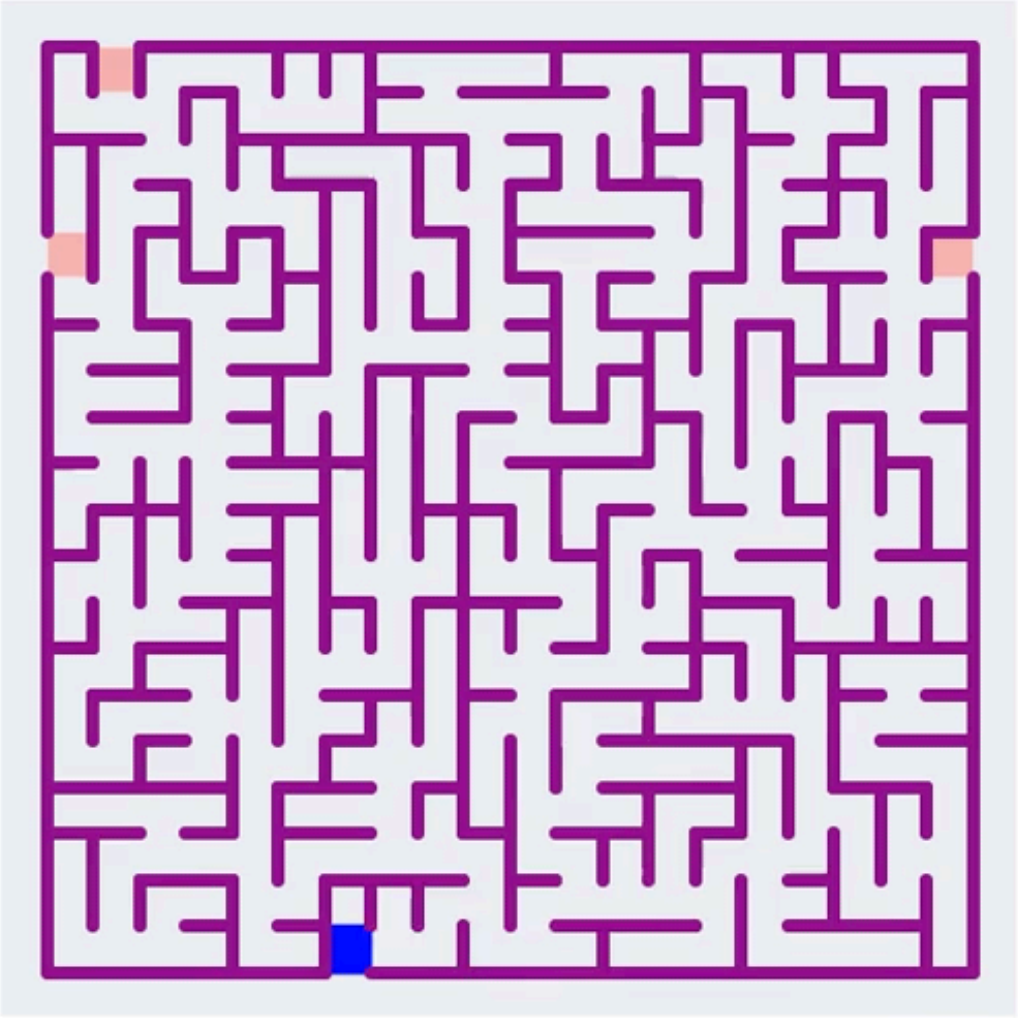
generate_subset()的递归树



红色数字代表函数调用顺序

回溯

- 构造所有子集
- 构造所有排列
- 构造所有路径
- n皇后
- 子集和
- 文本分割
- 最长递增子序列
- 最优二叉搜索树



构造所有排列

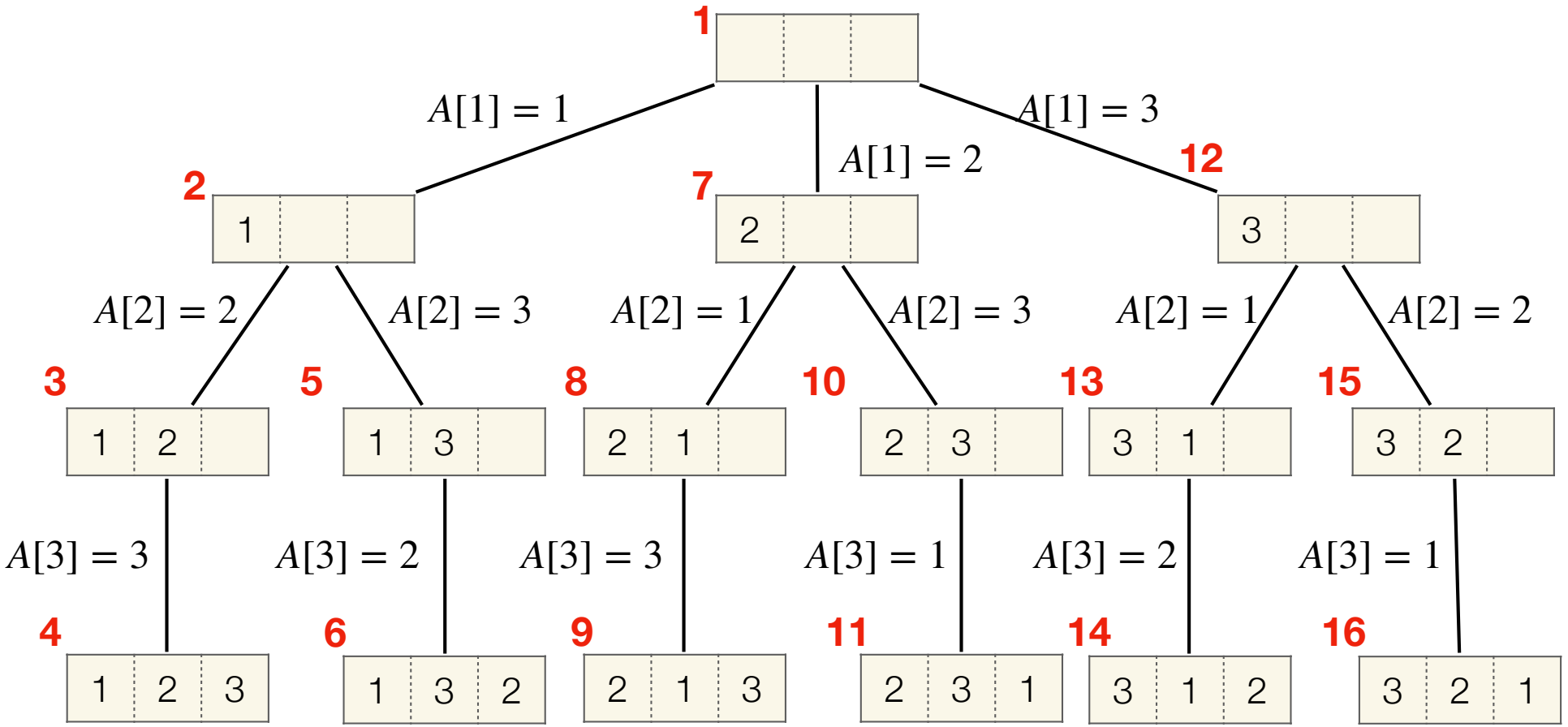
- 如何构造$1,2,\cdots,n$的所有排列
- 构造一个排列可以看成是一个长度为n的决策序列
  - 第i步决定将哪个数放入排列的第i个位置
    - $n-i+1$种候选方案，具体值取决于前$i-1$步的决策
    - 如何快速判断数字$c \in \{1,2,\cdots,n\}$属于候选方案？
      - 全局数组used$[1..n]$，used$[c]$表示数字c在之前的决策$A[1..i-1]$中是否已经使用
  - 每个长度为n的决策序列对应一个排列

算法实现

- 全局数组$A[1..n]$存放所有的决策
- 全局数组used$[1..n]$用来快速确定每一步的候选方案
- $i > n$时，排列的n个位置上都有确定的数字，找到一个解

```
def generate_permutation(i):
    if i > n:
        print(A[1:])
    else:
        for c in range(1, n + 1):
            if not used[c]:
                used[c] = True
                A[i] = c
                generate_permutation(i + 1)
                used[c] = False

n = 4
A = [None] * (n + 1)
used = [None] * (n + 1)
generate_permutation(1)
```

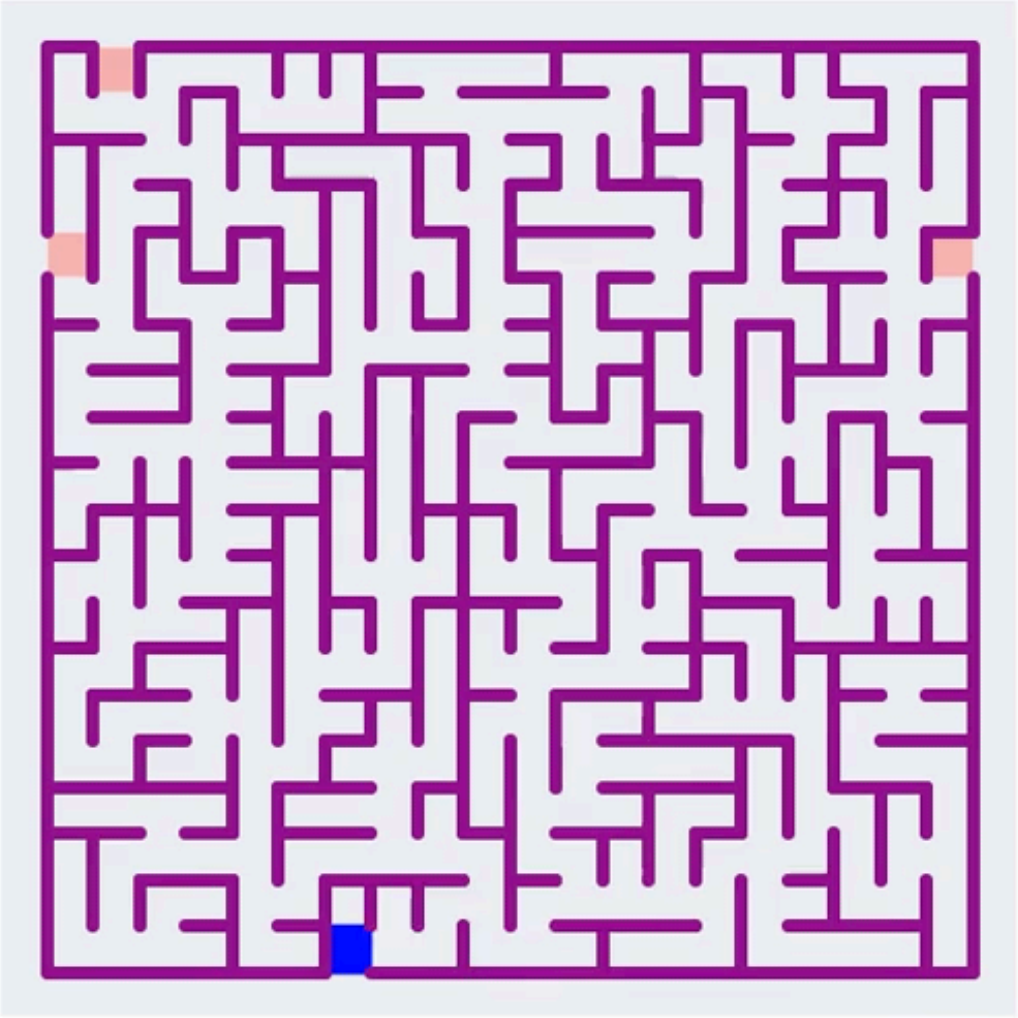
generate_permutation()的递归树



红色数字代表函数调用顺序

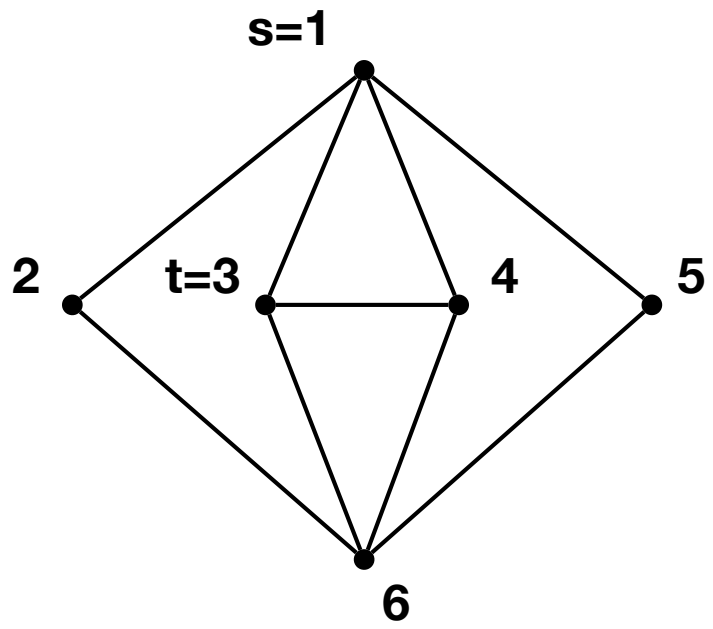
回溯

- 构造所有子集
- 构造所有排列
- 构造所有路径
- n皇后
- 子集和
- 文本分割
- 最长递增子序列
- 最优二叉搜索树



构造所有路径

- 如何在给定图$G = (V, E)$中构造所有从s到t的简单路径
  - 简单路径中所有顶点互不相同



$p_1 = 1,3$

$p_2 = 1,4,3$

$p_3 = 1,4,6,3$

$p_4 = 1,2,6,3$

$p_5 = 1,2,6,4,3$

$p_6 = 1,5,6,3$

$p_7 = 1,5,6,4,3$

构造所有路径

- 如何在给定图$G=(V,E)$中构造所有从s到t的简单路径
  - 简单路径中所有顶点互不相同
- 一条路径可以看成一个长度最多为$n=|V|$的决策序列
  - 第i步决定将哪个顶点放入路径的第i个位置
    - 第1步只有一种候选方案，即将顶点s加入路径
    - 第2步的候选方案包括s的所有邻接点
    - 第i步的候选方案取决于第$i-1$步的决策$A[i-1]$，即它的所有未放入当前路径的邻接点
    - 如果$A[i-1]=t$，那么已经找到s到t的简单路径
  - 每个长度为$n \le |V|$的决策序列对应一条长度为n的路径

算法实现

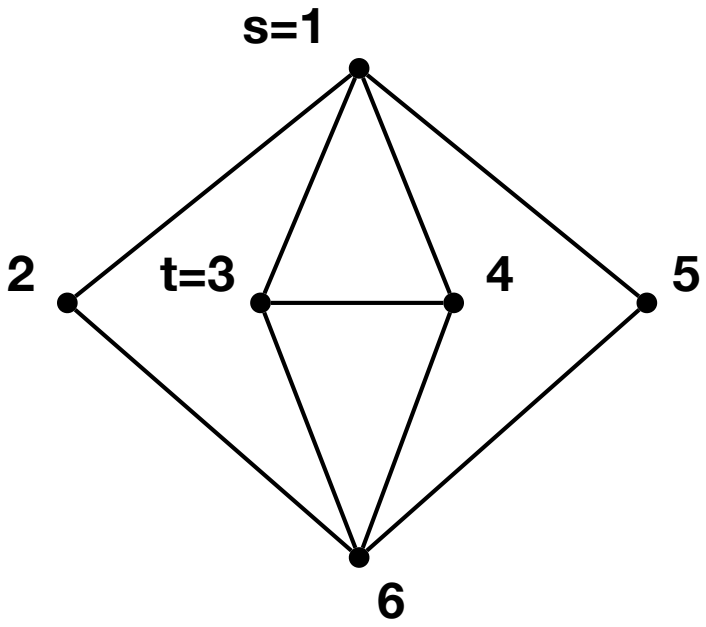
- 全局数组$A[1..n]$存放所有的决策
- 全局数组used$[1..n]$用来快速确定每一步的候选方案
- $A[i-1]=t$时，找到一条从s到t的路径，即$A[1..i-1]$

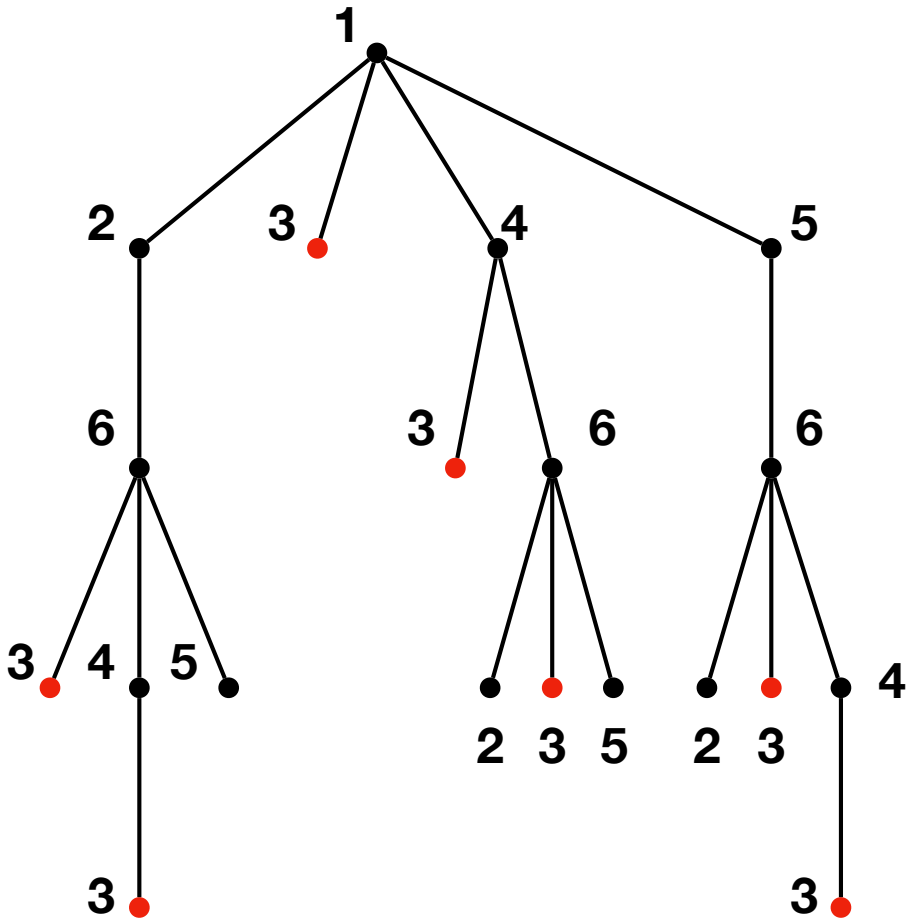
```
def generate_path(i):
    if i == 1:
        A[i] = s
        used[s] = True
        generate_path(i+1)
    else:
        if A[i-1] == t:
            print(A[1:i])
        else:
            for c in g[A[i-1]]:
                if not used[c]:
                    used[c] = True
                    A[i] = c
                    generate_path(i+1)
                    used[c] = False
```

```
g = {1:{2,3,4,5}, 2:{1,6},
     3:{1,4,6}, 4:{1,3,6},
     5:{1,6}, 6:{2,3,4,5}}
s = 1
t = 3
n = len(g)
A = [None] * (n + 1)
used = [False] * (n + 1)
generate_path(1)
```

generate_path()的递归树

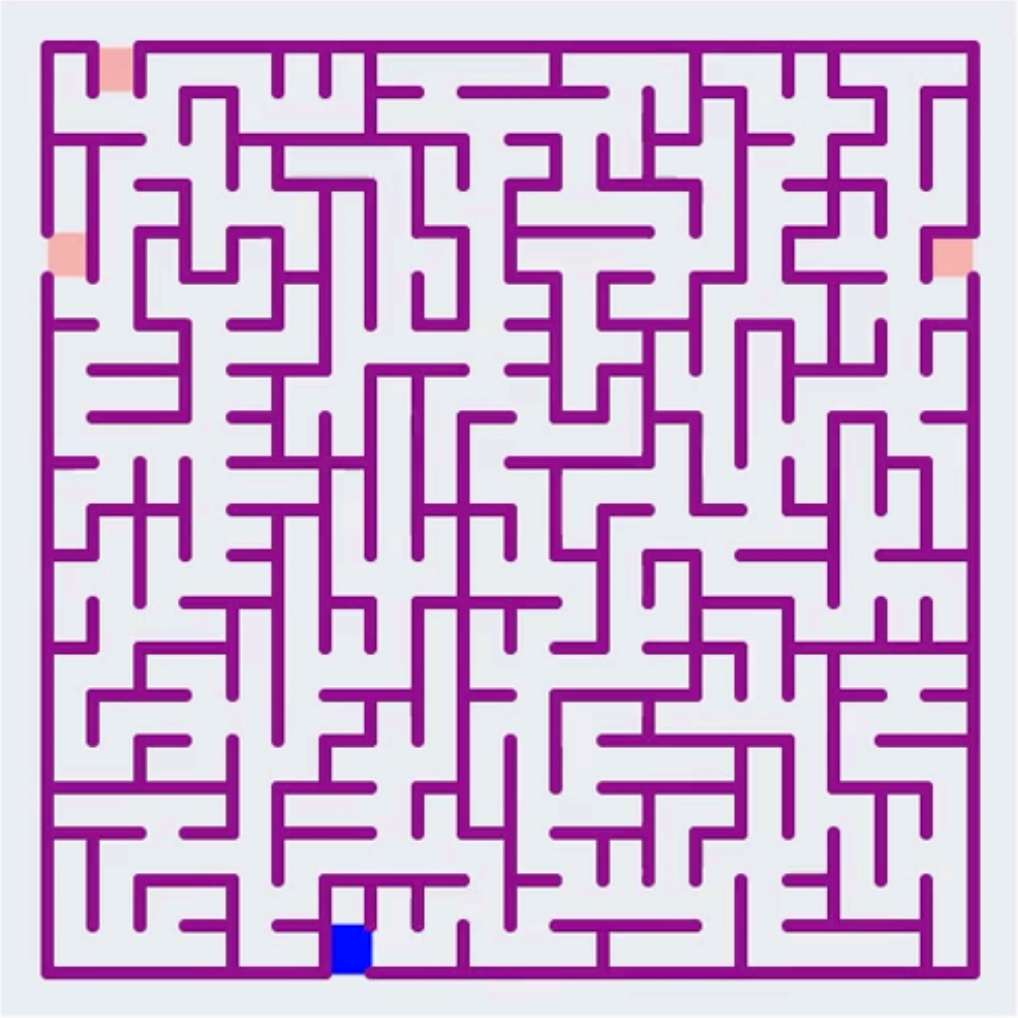
- 如何在给定图$G=(V,E)$中构造所有从s到t的简单路径
  - 简单路径中所有顶点互不相同





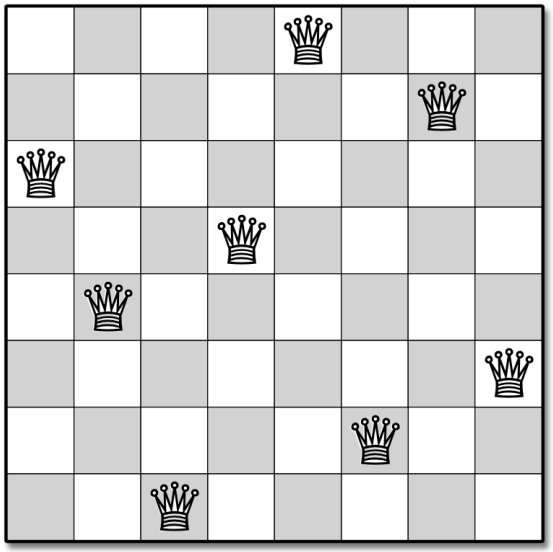
回溯

- 构造所有子集
- 构造所有排列
- 构造所有路径
- n皇后
- 子集和
- 文本分割
- 最长递增子序列
- 最优二叉搜索树



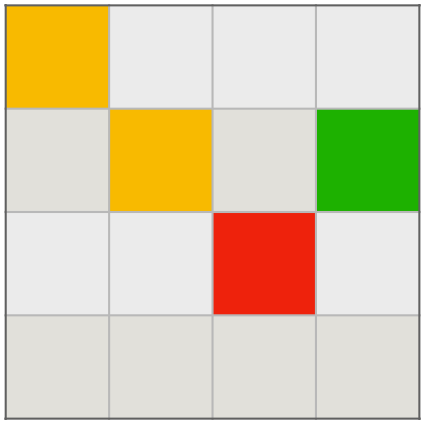
n皇后

- 如何将n个皇后放置在一个$n \times n$的棋盘上，使得任意两个皇后都不在同一行或同一列或同一对角线上
- 穷举法（8皇后）
  - 没有2个在皇后在同一位置：$\binom{64}{8} = 4426165368$
  - 没有2个皇后在同一行：$8^8 = 16777216$
  - 没有2个皇后在同一行或同一列：$8! = 40320$
- 回溯
  - 8皇后：2057次递归调用
  - 4皇后：17次递归调用



回溯法求解n皇后

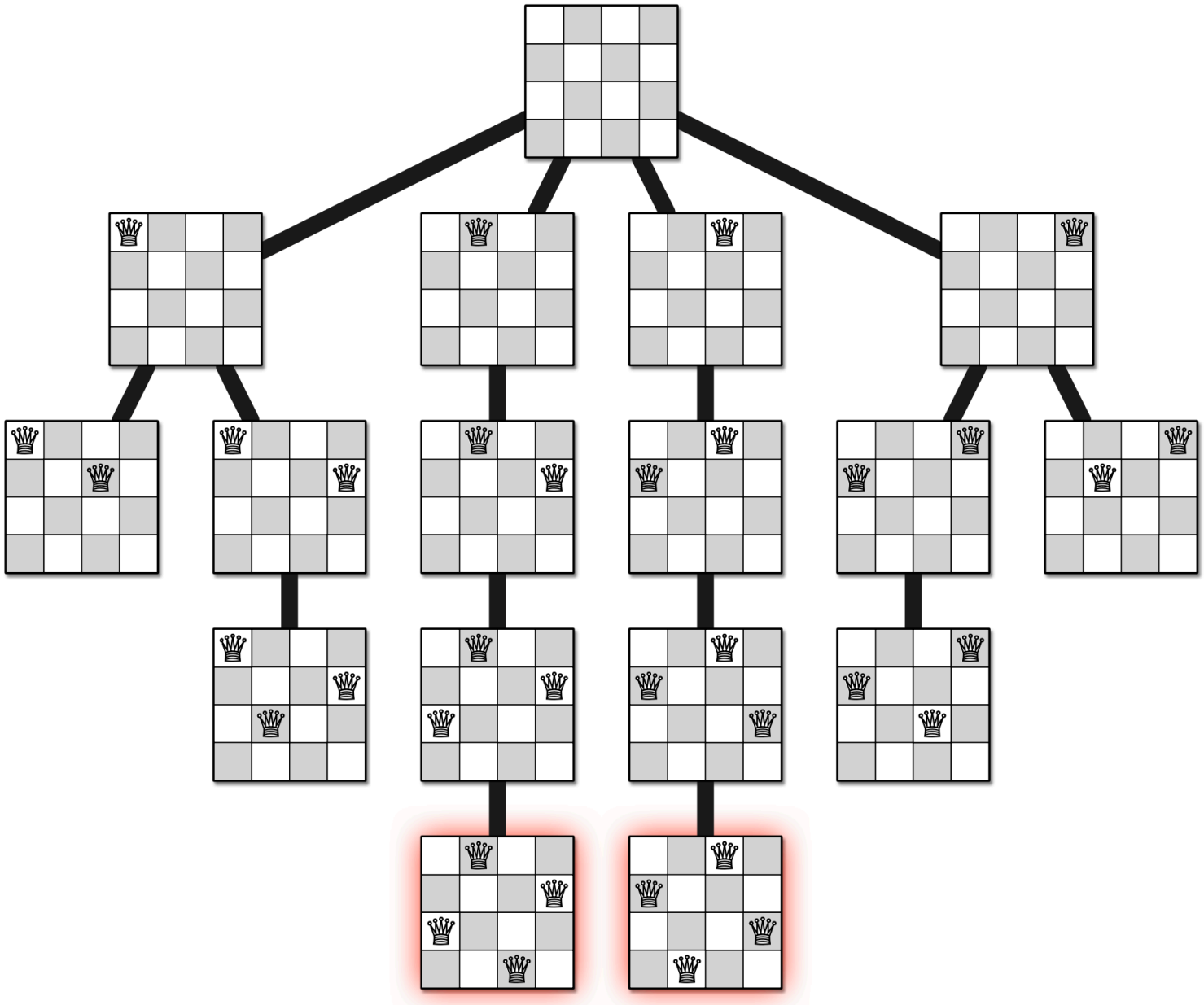
- n皇后的一个解可以看成一个长度为n的决策序列
  - 第i步决定将第i个皇后放在第i行的哪一列上
    - 第i步的候选方案取决于前$i-1$步的决策$A[1..i-1]$
      - $A[j]$表示第j个皇后所在的列数（其所在行数是j）
      - 对于列c，如果存在一个皇后j，$1 \le j \le i-1$，使得$A[j]=c$或$|A[j]-c|=|j-i|$，那么它不是候选方案
  - 每个长度为n的决策序列对应一个解



算法实现

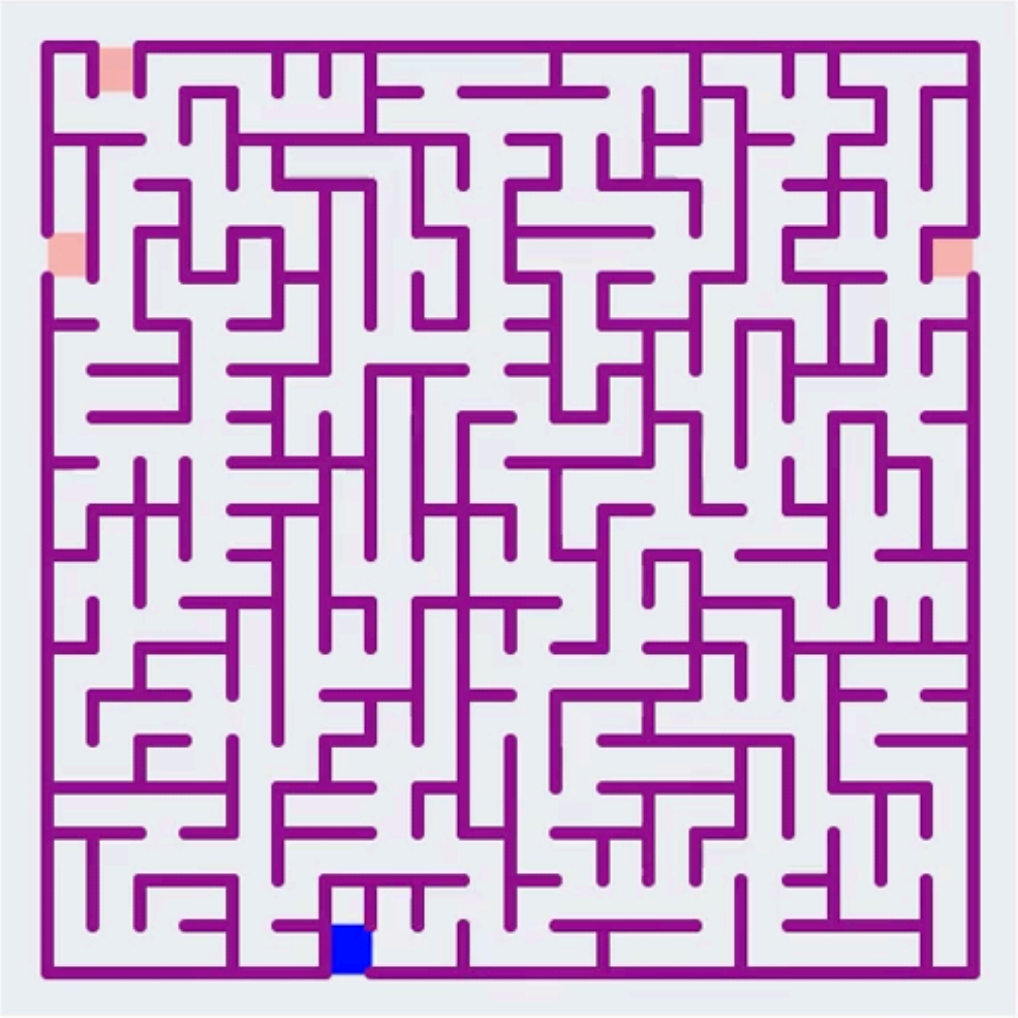
```
def place_queens(i):
    if i > n:
        print(A[1:])
    else:
        for c in range(1, n + 1):
            legal = True
            for j in range(1, i):
                if A[j] == c or abs(A[j] - c) == abs(j - i):
                    legal = False
            if legal:
                A[i] = c
                place_queens(i + 1)

n = 8
A = [None] * (n + 1)
place_queens(1)
```

4皇后问题的递归树



回溯

- 构造所有子集
- 构造所有排列
- 构造所有路径
- n皇后
- 子集和
- 文本分割
- 最长递增子序列
- 最优二叉搜索树



子集和

- 令X是一个包含n个正整数的集合，t是一个正整数
- 是否存在X的一个子集Y，其中所有元素的和等于t
  - 如果存在，返回True；否则返回False
  - 比如$X = \{8,6,7,5,3,10,9\}$, $t = 15$时，返回True
    - 集合$\{8,7\}$, $\{7,5,3\}$, $\{6,9\}$, $\{10,5\}$都满足条件
  - 比如$X = \{11,6,5,1,7,13,12\}$, $t = 15$时，返回False
- 假设存在子集Y，其中所有元素的和等于t，对于$\forall y \in Y$
  - $Y \backslash \{y\}$必然有一个子集，其中所有元素的和等于$t - y$

回溯法求解子集和

- 子集和的一个解可以看成一个长度不超过n的决策序列
- 第1步：$X[1..n]$中是否存在一个子集，其所有元素的和等于t
- 第i步：$X[i..n]$中是否存在一个子集，其所有元素的和等于d
  - 决定是否将元素$X[i]$放入子集Y（只有如下两种方案，与之前的决策无关）
    - 不放：$X[i+1..n]$中是否存在一个子集，其所有元素的和等于d
    - 放：$X[i+1..n]$中是否存在一个子集，其所有元素的和等于$d-X[i]$
  - 如何判断$A[1..i-1]$是否是一个解
    - $d=0$：是一个解，因为$\varnothing$是$X[i..n]$的一个子集，其所有元素的和等于0
    - $d<0$：不是一个解，且没有必要扩展（考虑是否放入$X[i]$）
    - $d>0$
      - 如果$i>n$，当前决策序列不是一个解
      - 否则，考虑是否将元素$X[i]$放入子集Y

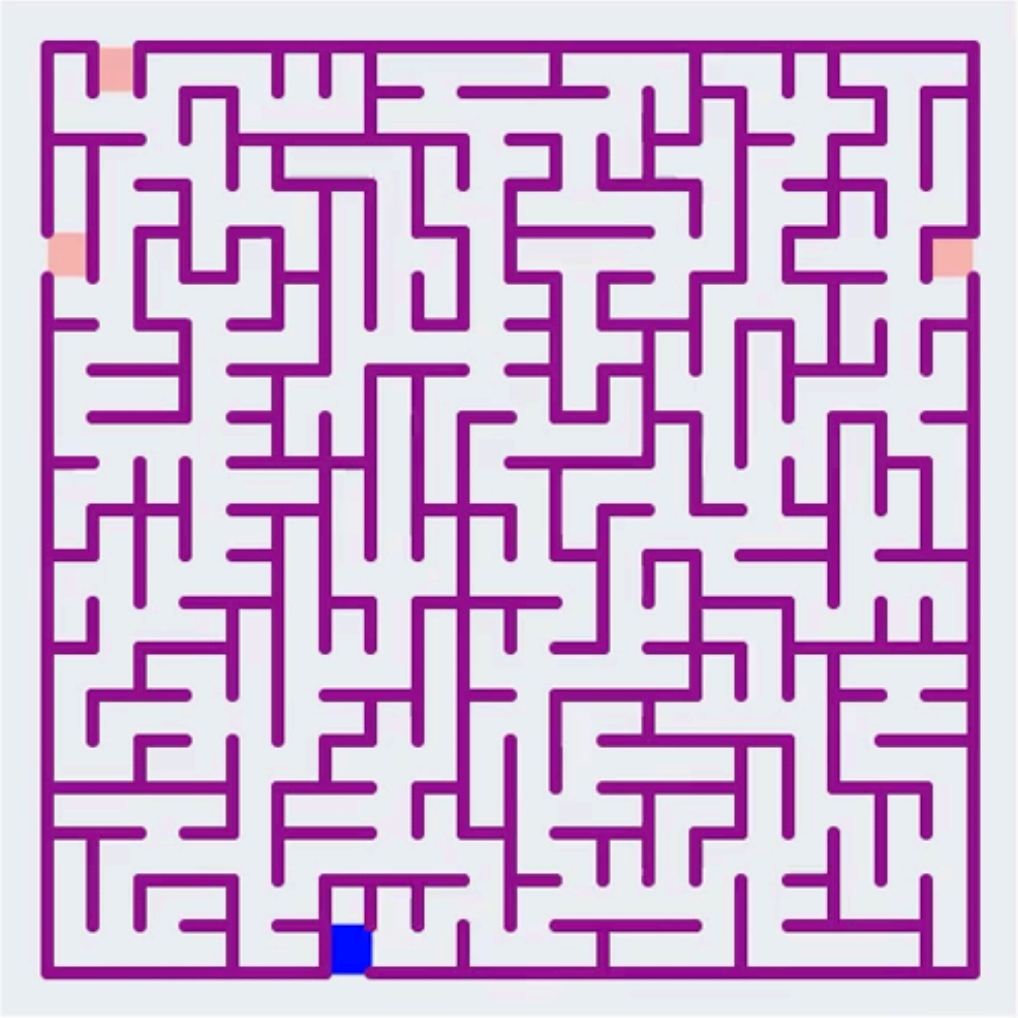
算法实现

```
def generate_subset_sum(i, d):
    if d == 0:
        print(A[1:i])
    elif d > 0:
        if i <= n:
            for c in range(2):
                A[i] = c
                if c == 0: #skip
                    generate_subset_sum(i+1, d)
                else: #take
                    generate_subset_sum(i+1, d-X[i])

X = [None, 8, 6, 7, 5, 3, 10, 9]
t = 15
A = [None] * len(X)
n = len(X) - 1
generate_subset_sum(1, t)
```

$$T(n) = 2T(n-1) + O(1) \Rightarrow T(n) = O(2^n)$$

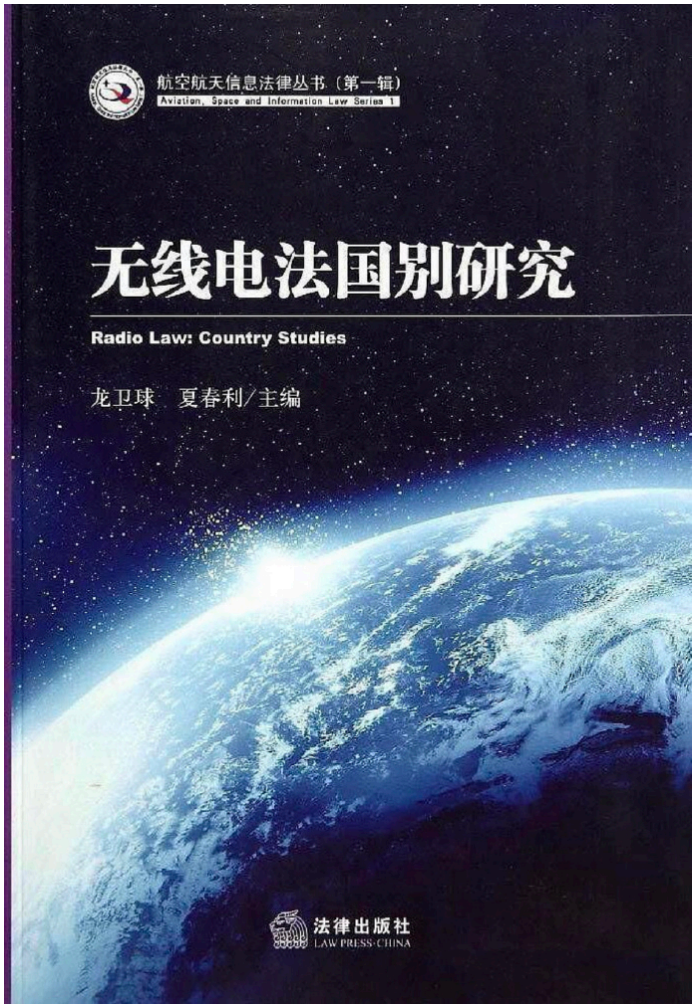
回溯

- 构造所有子集
- 构造所有排列
- 构造所有路径
- n皇后
- 子集和
- 文本分割
- 最长递增子序列
- 最优二叉搜索树



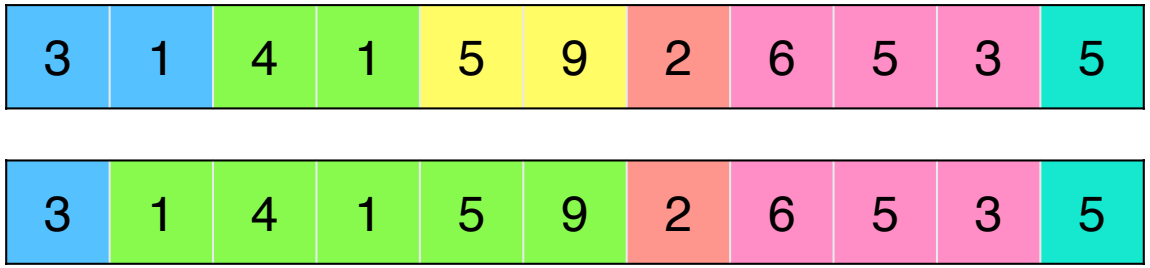
文本分割

- 给定一串字符，能否将其分割成若干单词？
- 比如：无线电法国别研究
  - 无线电 + 法国 + 别 + 研究
  - 无线电 + 法 + 国别 + 研究



文本分割

- 辅助函数is_word(w)
  - 输入：w是一个字符串
  - 输出：如果w是一个单词，返回True，否则返回False
  - 如果定义回文串是单词
    - is_word(refer)返回True，is_word(bad)返回False
  - 如果定义质数是单词
    - is_word(31)返回True，is_word(314)返回False
- 问题：31415926535能否分割成一个质数单词序列？



回溯法求解文本分割

- 文本分割的一个解可以看成一个长度不超过n的决策序列
- 第1步确定第1个单词w_1
  - 候选方案：$\{s[1..j] \mid is\_word(s[1..j]) = True,\ 1 \le j \le n\}$
- 第i步确定第i个单词w_i
  - w_i的起始位置取决于第$i-1$步确定的单词w_{i-1}的结束位置
    - 将w_i的结束位置放入决策数组$A[i]$
  - 候选方案：
    $\{s[A[i-1]+1..j] \mid is\_word(s[A[i-1]+1..j]) = True,\ A[i-1]+1 \le j \le n\}$
- 如何判断$A[1..i-1]$是否是一个解
  - $A[i-1] = n$：单词w_{i-1}已经包含最后一个字符，找到一个解
  - 否则，确定第i个单词w_i

算法实现

```
def generate_segmentation(i):
    if A[i-1] == n:
        print(A[1:i])
    else:
        for c in range(A[i-1]+1, n+1):
            if is_word(s[A[i-1]+1 : c+1]):
                A[i] = c
                generate_segmentation(i+1)
```

```
s = ' 31415926535'
n = len(s) - 1
A = [0] * n
generate_segmentation(1)
```

- $T(n)$：算法generate_segmentation(n)中函数is_word的调用次数

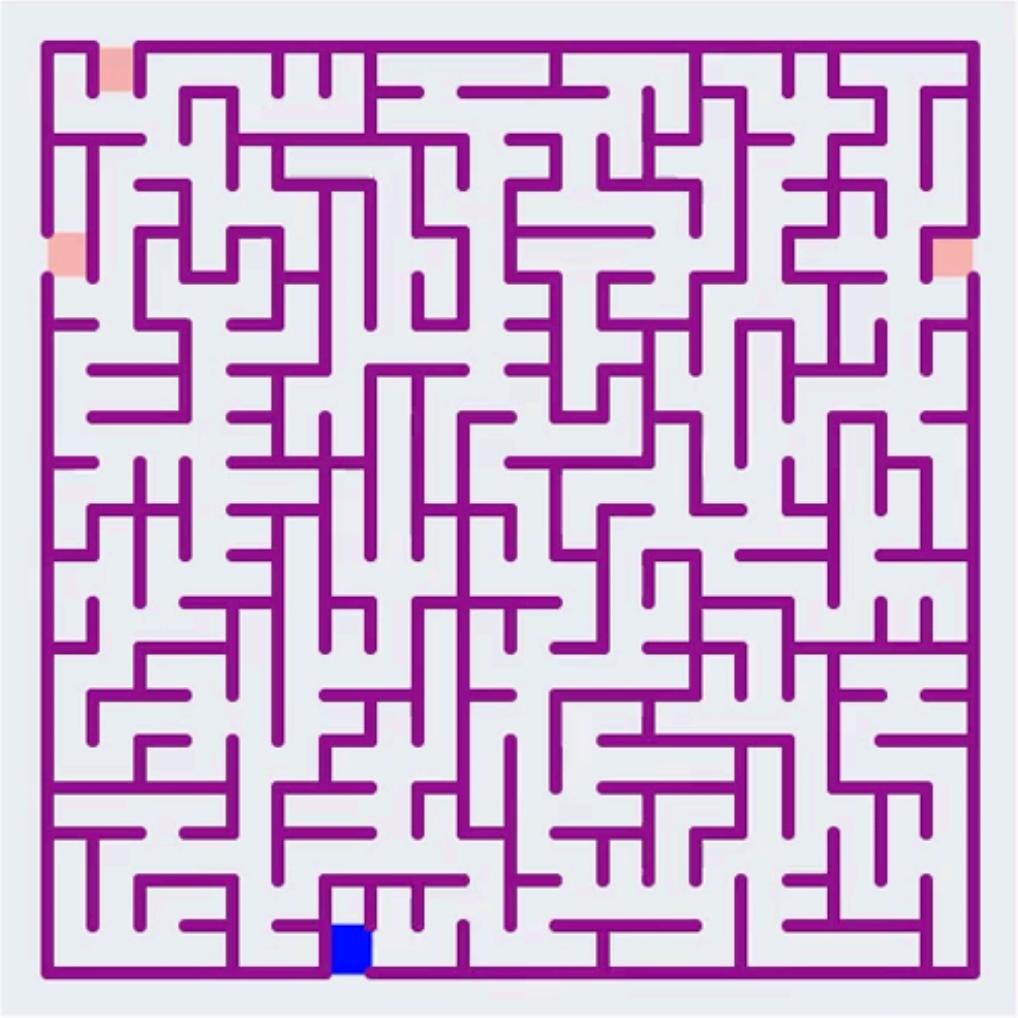
$$T(n) \leq \sum_{i=0}^{n-1} T(i) + n$$

- 假设对于任意的i和j，is_word(i, j)均返回True

$$\left.\begin{aligned} T(n) &= \sum_{i=0}^{n-1} T(i) + n \\ T(n-1) &= \sum_{i=0}^{n-2} T(i) + (n-1) \end{aligned}\right\} \begin{aligned} &\Rightarrow T(n) - T(n-1) = T(n-1) + 1 \\ &\Rightarrow T(n) = 2T(n-1) + 1 \quad \Rightarrow T(n) = O(2^n) \end{aligned}$$

回溯

- 构造所有子集
- 构造所有排列
- 构造所有路径
- n皇后
- 子集和
- 文本分割
- 最长递增子序列
- 最优二叉搜索树



最长递增子序列

- 子序列：对于任意序列S，它的子序列是通过删除其中零个或多个元素得到的另一个序列（注意：剩余元素的顺序并不改变）
  - backtracking的子序列包括：bring, tracking, 空串， backtracking
- 子串：如果一个子序列的所有元素在原始串中是连续的，那么它就是一个子串，比如tracking
- 给定一个整数序列，找出其中最长的递增子序列

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

回溯法求解最长递增子序列

- 最长递增子序列的一个解可以看成一个长度为n的决策序列
- 第i步：决定是否将元素$X[i]$放入子序列
  - 不放，$A[i] = 0$：始终可行的决策，和之前的决策无关
  - 放，$A[i] = 1$：该决策是否可行取决于最后一个放入子序列的元素t，如果$X[i] > t$，可以放入，并更新t的值
    - 令t的初始值为$-\infty$，便于第1步处理
- 如何判断$A[1..i-1]$是否是一个解
  - 当$i > n$时，找到一个新的无法扩展的递增子序列，如果其长度大于之前找到的递增子序列，更新目前找到的解

算法实现

```
def generate_lis(i, t, l): # l: length of current increasing list
    global m #length of longest increasing list so far
    if i > n:
        if l > m: # find a new longer increasing list
            sol.clear()
            sol.add(tuple(A[1:]))
            m = l
        elif l == m: # find another list that is longest so far
            sol.add(tuple(A[1:]))
    else:
        for c in range(2): #
            if c == 0: # skip
                A[i] = 0
                generate_lis(i+1, t, l)
            else: # want to take
                if X[i] > t:
                    A[i] = 1
                    generate_lis(i+1, X[i], l+1)

X = [None, 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8]
A = [None] * len(X)
n = len(X) - 1
m = 0
sol = set()
generate_lis(1, float('-inf'), 0)
print(sol)
```

另一种决策方案

- 最长递增子序列的一个解可以看成一个长度不超过n的决策序列
- 第i步：决定选择哪个元素放入子序列的第i个位置
  - 取决于最后一个放入子序列的元素$X[j]$
  - 候选方案：$\{X[k] \mid k > j \text{ and } X[k] > X[j]\}$
  - 记录所选元素的索引$k: A[i] = k$
  - 候选方案可能为空，意味着目前的子序列已经不能再扩展，决策序列到此结束
    - X的最后一个元素已经放入子序列
    - 剩余元素均小于$X[j]$

算法实现

```
def generate_lis_1(i):
    global m
    candidates = [c for c in range(A[i-1]+1, n+1) if X[c] > X[A[i-1]]]
    if not candidates:
        if i-1 > m: #i-1 is the length of current increasing list
            sol.clear()
            sol.add(tuple(A[1:i]))
            m = i-1
        elif i-1 == m:
            sol.add(tuple(A[1:i]))
    else:
        for c in candidates:
            A[i] = c
            generate_lis_1(i+1)

X = [float('-inf'), 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
A = [0] * len(X)
n = len(X) - 1
m = 0
sol = set()
generate_lis_1(1)
print(sol)
```

两种回溯算法的效率比较

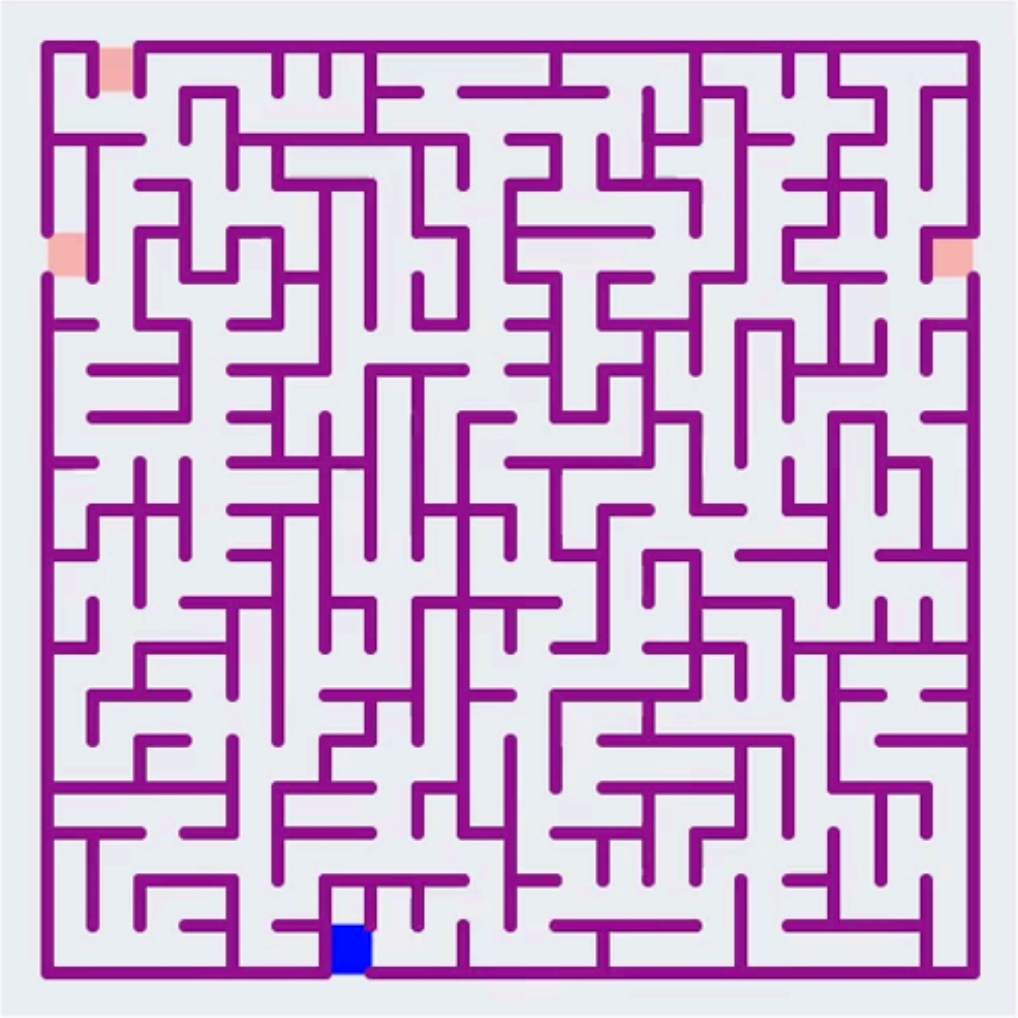
算法	generate_lis( )	generate_lis_1( )
策略	第i步决定是否将元素X[i]放入子序列	第i步决定选择哪个元素放入子序列的第i个位置
候选方案数量	O(1)	O(n)
递归调用次数	多	少

两种算法递归调用次数的比较

		generate_lis( )	generate_lis_1( )	
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8]		488	146	30%
[100, 99, …, 2, 1]		5151	101	2%
大小为30的随机列表	第1次实验	16474	2797	17%
	第2次实验	11792	2045	17%
	第3次实验	17658	3594	20%
	第4次实验	12954	2019	16%
	第5次实验	30256	4785	16%

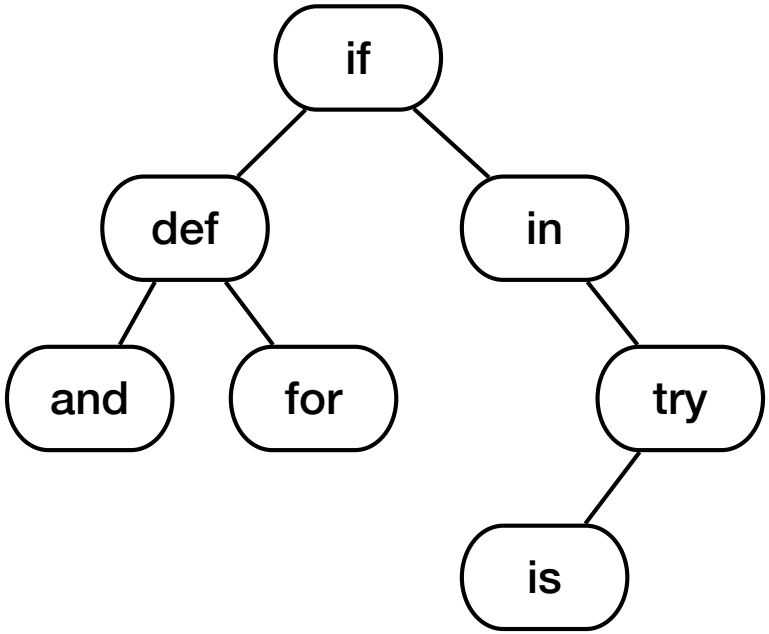
回溯

- 构造所有子集
- 构造所有排列
- 构造所有路径
- n皇后
- 子集和
- 文本分割
- 最长递增子序列
- 最优二叉搜索树



二叉搜索树

- 二叉搜索树是指一棵空树或者具有下列性质的二叉树
  - 若左子树不空，则其所有节点的值均小于根节点的值
  - 若右子树不空，则其所有节点的值均大于根节点的值
  - 左子树和右子树都是二叉搜索树

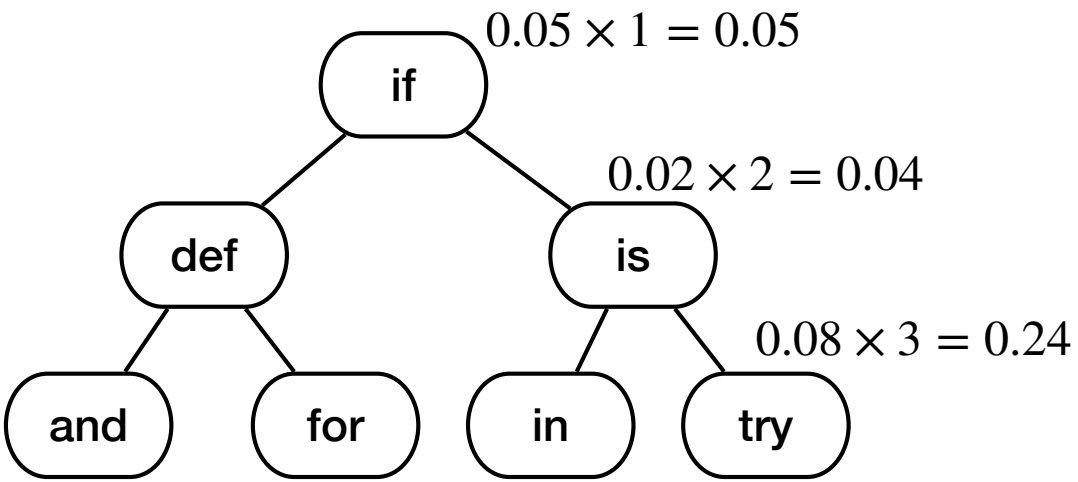


最优二叉搜索树

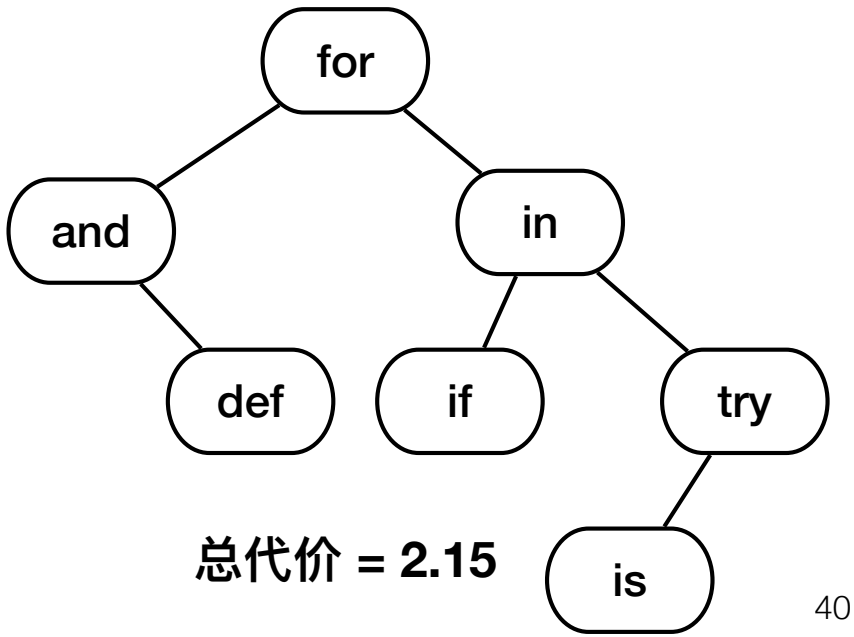
- 给定n个节点$v_1, \cdots, v_n$，它们具有不同的搜索概率：$f[1..n]$

and	def	for	if	in	is	try
0.22	0.18	0.20	0.05	0.25	0.02	0.08

- 节点v_i的搜索代价$c(v_i) = d(v_i) + 1$
- 二叉搜索树T的总搜索代价$\text{Cost}(T, f[1..n]) = \sum_{i=1}^{n} f[i] \cdot c(v_i)$



总代价 = 2.7



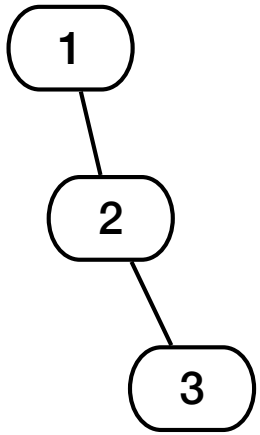
总代价 = 2.15

构造所有二叉搜索树

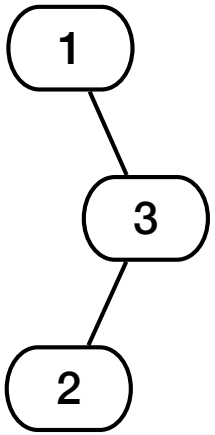
- 一棵二叉搜索树可以看成一个长度为n的决策序列
- 第1步：从$V[1..n]$中选择一个节点v_k作为根节点，其左子树的节点集合是$V[1..k-1]$，右子树的节点集合是$V[k+1..n]$
  - 显然有n种候选方案
- 第i步：从$V[i..j]$中选择一个节点
- 回溯 + 分治
  - 问题：给定n个节点$V[1..n]$，构造所有的二叉搜索树
  - 分解：从$V[1..n]$中选择一个节点v_k作为根节点，其左子树的节点集合是$V[1..k-1]$，右子树的节点集合是$V[k+1..n]$
  - 解决：递归求解$V[1..k-1]$可以构造的二叉搜索树L以及$V[k+1..n]$可以构造的二叉搜索树R
  - 合并：v_k + L中的每一棵树 × R中的每一棵树

算法实现

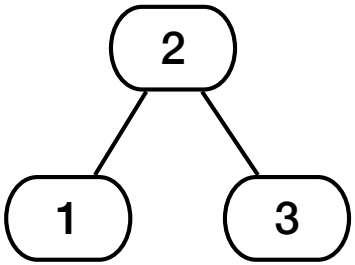
```
def generate_bst(low, high):
    if low > high:
        return [[]] #an empty tree
    elif low == high:
        return [[low]] #a tree with only one node
    else:
        bst = []
        for c in range(low, high + 1):
            L = generate_bst(low, c-1)
            R = generate_bst(c+1, high)
            ts = [[c, l, r] for l in L for r in R]
            bst.extend(ts)
        return bst # a list of all possible trees
```



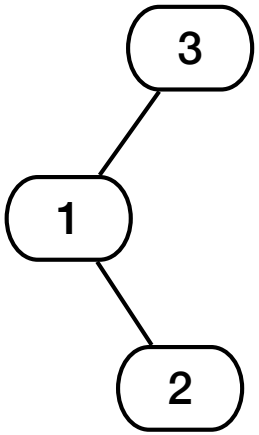
[1, [ ], [2, [ ], [3]]]



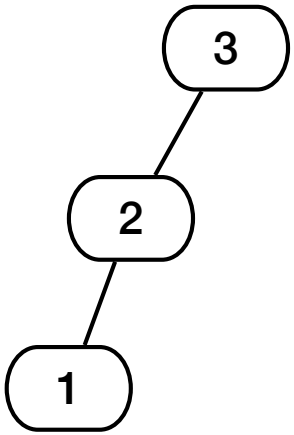
[1, [ ], [3, [2], [ ]]]



[2, [1], [3]]



[3, [1, [ ], [2]], [ ]]



[3, [2, [1], [ ]], [ ]]

最优二叉搜索树

- 通过回溯+分治构造出所有的二叉搜索树
- 遍历求出最优二叉搜索树
- 时间复杂度取决于具有n个节点的二叉搜索树的数量
- $T_0 = 0,\ T_1 = 1,\ T_2 = 2,\ T_3 = 5$

$$T_n = \sum_{1 \le k \le n} T_{k-1} T_{n-k}$$

$$\Rightarrow T_n = \frac{1}{n+1}\binom{2n}{n}$$ ← Catalan数