中国科学技术大学
University of Science and Technology of China

实用算法设计

➢ 查找

主讲：娄文启

目录

1. 递归查找（回溯）-01背包问题
2. 基于散列表的查找：按内容的查找
   - 可结合位图、 BloomFilter这些数据结构来灵活实现

假定：待查找的数据已在内存中

重点：理解各类查找算法的特点及适用场景。
难点：针对具体应用合理选择数据结构及查找算法。

基础：C语言编程；顺序表、链表和hash表的定义及基本操作的实现。

查找的广泛应用

“按值”查找

经典场景：

- 编译器查询变量名以得到其类型和地址；符号表
- 拼写检查器查字典以确保单词拼写正确；
- 电话号码簿程序查询用户名以找到其电话号码；
- 因特网域名服务器查找域名来发现IP地址；DNS

关键问题：

1. 如何在内存中存储一组数据？
2. 如何查询？（查询算法的选择/设计）

查找的特征

- 准备时间:
  - 比如，二分查找算法必须在有序表上运用才有效，因此，必须要先对一个无序表进行排序，才能使用二分查找
  - 准备时间可能会超过查找本身所节约的时间。
    - 蛮力查找（顺序查找） vs. 二分查找

排序的时间-准备时间

- 运行时间：(N表示被查找对象的规模)
  - $O(1) < O(\lg N) < O(N) < O(N*\lg N) < O(N^2) < O(N^k) < O(2^N)$
- 回溯的需要:
  - 例1：0-1背包问题：http://blog.csdn.net/ljc_zy/article/details/629418

    给定n种物品和一背包。物品i的重量是wi，其价值为vi，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?
  - 例2：N皇后问题
    https://blog.csdn.net/piyongduo3393/article/details/86497081

0-1背包问题

问题定义

● 形式化定义

- 输入：
  - N 个商品组成集合 O，每个商品有两个属性 v_i 和 p_i，分别表示体积和价格
  - 背包容量为 V

- 输出：
  - 求解一个商品子集 $S \subseteq O$，令

$$\max \sum_{i \in S} p_i \quad \text{（优化目标）}$$

  - 满足约束条件：

$$\sum_{i \in S} v_i \leqslant V \quad \text{（约束条件）}$$

0-1背包问题（递归）

蛮力枚举

KnapsackSR(i, V): 前 i 个商品中，容量为 V 时最优解

算法 2.1　背包问题的回溯

```
1  输入: 前 i 个商品, 背包容量 V;
2  输出: 最大总价格 P;
3  if V < 0 then
4      return −∞
5  else
6      if i ⩽ 0 then
7          return 0
8      else
9          P_1 ← KnapsackSR(i − 1, V − v_i);
10         P_2 ← KnapsackSR(i − 1, v);
11         P ← max(P_1 + p_i, P_2);
12         return P
```

0-1背包问题（递归）

蛮力枚举

KnapsackSR(i, V): 前 i 个商品中，容量为 V 时最优解

算法 2.1　背包问题的回溯

```
1  输入: 前 i 个商品, 背包容量 V;
2  输出: 最大总价格 P;
3  if V < 0 then
4      return −∞
5  else
6      if i ⩽ 0 then
7          return 0
8      else
9          P_1 ← KnapsackSR(i − 1, V − v_i);
10         P_2 ← KnapsackSR(i − 1, v);
11         P ← max(P_1 + p_i, P_2);
12         return P
```

退出条件1：超出背包容量

0-1背包问题（递归）

蛮力枚举　　KnapsackSR(i, V): 前 i 个商品中，容量为 V 时最优解

算法 2.1　背包问题的回溯

```
1  输入: 前 i 个商品, 背包容量 V;
2  输出: 最大总价格 P;
3  if V < 0 then
4      return −∞
5  else
6      if i ⩽ 0 then
7          return 0
8      else
9          P_1 ← KnapsackSR(i − 1, V − v_i);
10         P_2 ← KnapsackSR(i − 1, v);
11         P ← max(P_1 + p_i, P_2);
12         return P
```

退出条件2：商品决策完毕

0-1背包问题（递归）

蛮力枚举

KnapsackSR(i, V): 前 i 个商品中，容量为 V 时最优解

算法 2.1　背包问题的回溯

```
1  输入: 前 i 个商品, 背包容量 V;
2  输出: 最大总价格 P;
3  if V < 0 then
4      return −∞
5  else
6      if i ⩽ 0 then
7          return 0
8      else
9          P1 ← KnapsackSR(i − 1, V − vi);
10         P2 ← KnapsackSR(i − 1, v);
11         P ← max(P1 + pi, P2);
12         return P
```

递归解决子问题

0-1背包问题（递归）

蛮力枚举

KnapsackSR(i, V): 前 i 个商品中，容量为 V 时最优解

算法 2.1　背包问题的回溯

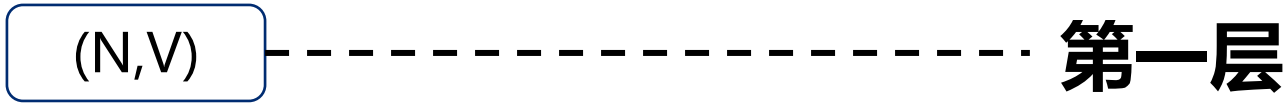
```
1  输入: 前 i 个商品, 背包容量 V;
2  输出: 最大总价格 P;
3  if V < 0 then
4      return −∞
5  else
6      if i ⩽ 0 then
7          return 0
8      else
9          P_1 ← KnapsackSR(i − 1, V − v_i);
10         P_2 ← KnapsackSR(i − 1, v);
11         P ← max(P_1 + p_i, P_2);
12         return P
```

确定最优子问题

0-1背包问题（递归）

蛮力枚举: 复杂度分析

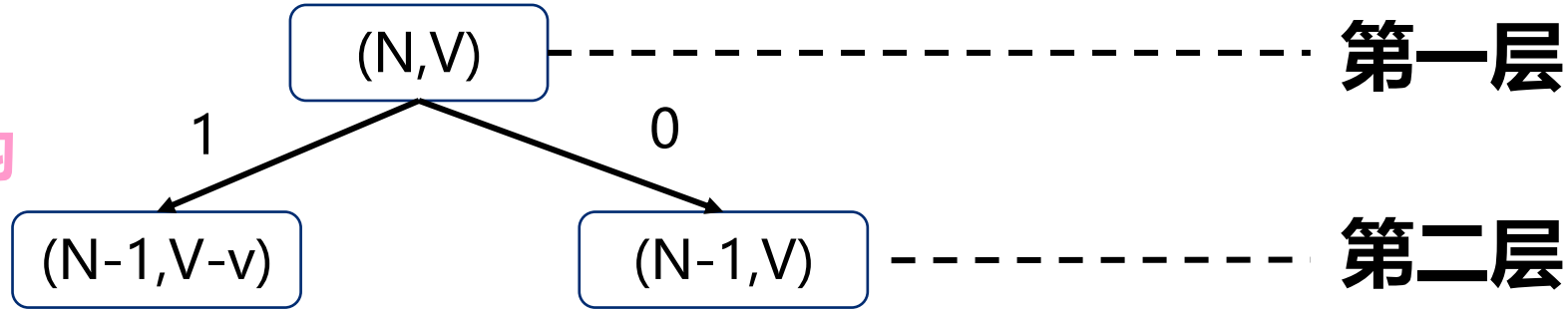
- 递归树
  - 设N个商品，体积均为v
  - 背包容量为V



0-1背包问题（递归）

蛮力枚举: 复杂度分析

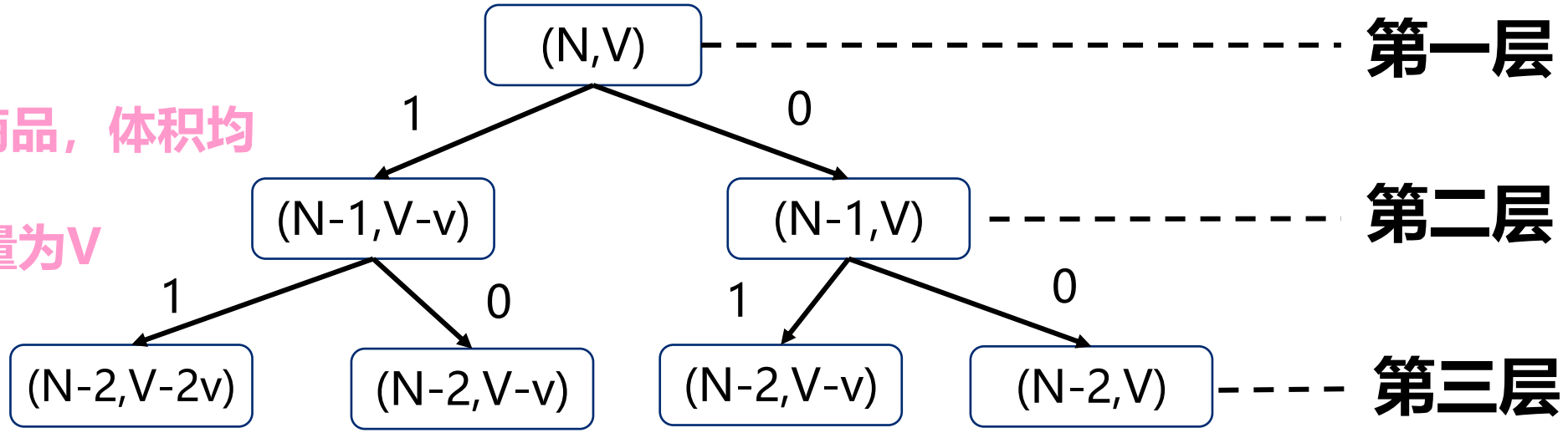
- 递归树
  - 设N个商品，体积均为v
  - 背包容量为V



0-1背包问题（递归）

蛮力枚举: 复杂度分析

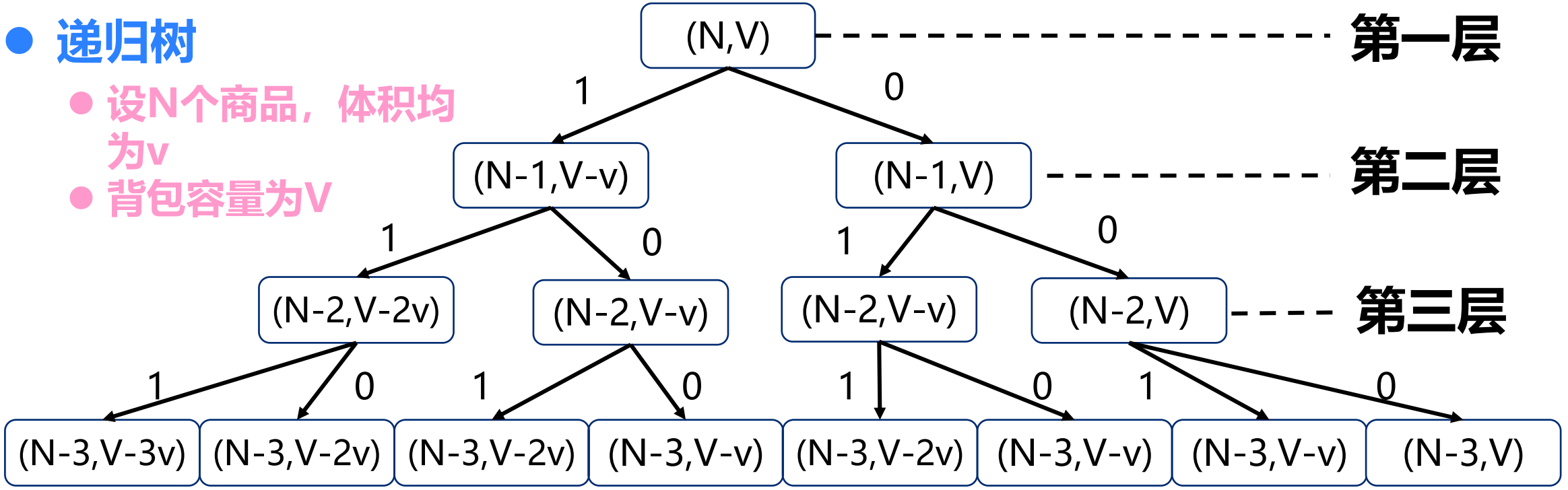
- 递归树
  - 设N个商品，体积均为v
  - 背包容量为V



0-1背包问题（递归）

蛮力枚举: 复杂度分析

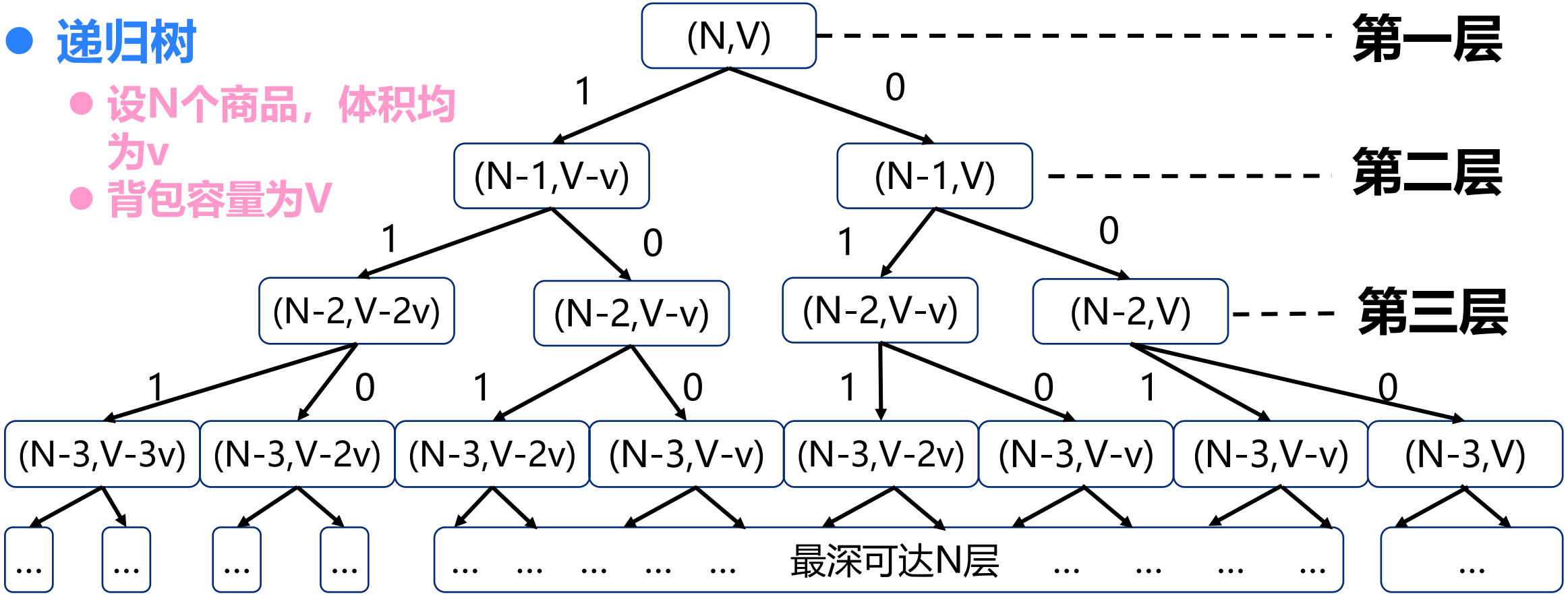
- 递归树
  - 设N个商品，体积均为v
  - 背包容量为V



0-1背包问题（递归）

蛮力枚举: 复杂度分析

- 递归树
  - 设N个商品，体积均为v
  - 背包容量为V



- 一棵深度为N的满二叉树

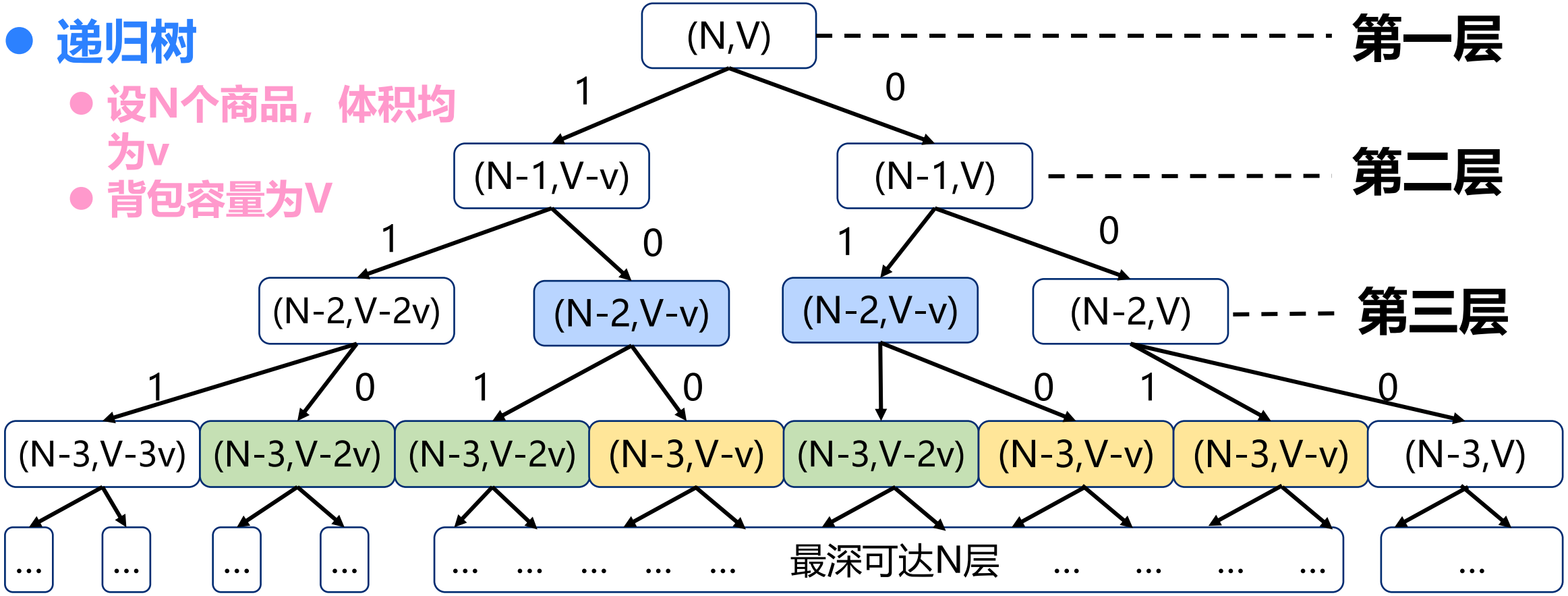
复杂度? $O(2^N)$

问题： 如何优化?

0-1背包问题（递归）

蛮力枚举: 复杂度分析

- 递归树
  - 设N个商品，体积均为v
  - 背包容量为V



- 大量重复子问题

剪枝 or 记录

0-1背包问题（递归优化-剪枝）

问题：如何安全地剪去一定不可能出现在最优解中的情况

物品属性如下：

	A	B	C	D
价值 Pi	60	100	160	40
体积vi	10	20	30	15

0-1背包问题（递归优化-剪枝）

问题：如何安全地剪去一定不可能出现在最优解中的情况

按照价值密度排序后：贪心

	A	C	B	D
价值 Pi	60	160	100	40
体积vi	10	30	20	15
密度	6	5.33	5	2.67

假设背包容量为50

贪心策略:A+C; 220
最优：C+B; 260

贪心不一定可以获得最优解

0-1背包问题（递归优化-剪枝）

问题：如何安全地剪去一定不可能出现在最优解中的情况

尽管贪心不保证最优解，但可以求的性能上界，用来剪枝：

算法 2.3　01 背包回溯剪枝

```
1  Function KnapsackBacktrack(i): ; //  回溯算法函数
2      if i > n then
3          update bestp; return
4      if cv + v[i] ⩽ c then
5          x[i] ← 1;
6          cv ← cv + v[i] ; cp ← cp + p[i];
7          KnapsackBacktrack(i + 1);
8          cv ← cv − v[i] ; cp ← cp − p[i];
9      if Bound(i + 1) > bestp then
10         x[i] ← 0;
11         KnapsackBacktrack(i + 1);
12     end
```

```
1  Function Bound(i): ; //  计算上界
2      cleft ← c − cv;
3      b ← cp;
4      while i ⩽ n and v[i] ⩽ cleft do
5          cleft ← cleft − v[i];
6          b ← b + p[i];
7          i ← i + 1;
8      end
9      if i ⩽ n then
10         b ← b + p[i]/v[i] × cleft;
11     end
       return b
```

0-1背包问题（递归优化-备忘录）

备忘录优化：伪代码

算法 2.2　带备忘录的回溯求解

```
1  输入: 商品集合 {1, ··· , i}，背包容量 V;
2  输出: 最大总价格 P[i, V];
3  if V < 0 then
4  |   return −∞;
5  if i ⩽ 0 then
6  |   return 0;

9  P_1 ← KnapsackMR(i − 1, V − v_i);
10 P_2 ← KnapsackMR(i − 1, V);
11 P[i, V] ← max(P_1 + p_i, P_2);
12 return P[i, V];
```

0-1背包问题（递归优化-备忘录）

备忘录优化：伪代码

算法 2.2　带备忘录的回溯求解

```
1  输入: 商品集合 {1, ··· , i}，背包容量 V;
2  输出: 最大总价格 P[i, V];
3  if V < 0 then
4  |   return −∞;
5  if i ⩽ 0 then
6  |   return 0;
7  if P[i, V] ≠ NULL then
8  |   return P[i, V];
9  P1 ← KnapsackMR(i − 1, V − vi);
10 P2 ← KnapsackMR(i − 1, V);
11 P[i, V] ← max(P1 + pi, P2);
12 return P[i, V];
```

重复子问题

构建备忘录P[i,V]
P[i,V]表示在前i个商品中选择，背包容量为V时的最优解

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

- 初始化
  - 容量为0：P[i,0] = 0
  - 没有商品时：P[0,V] = 0

P[i,V]	V=0	1	2	3	…	13	14	15	16
i=0	0	0	0	0	…	0	0	0	0
1	0								
…	0								
N	0								

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

- **确定计算顺序**
  - P[i-1,V-v_i]和P[i-1,V]位于P[i,V]的左上方

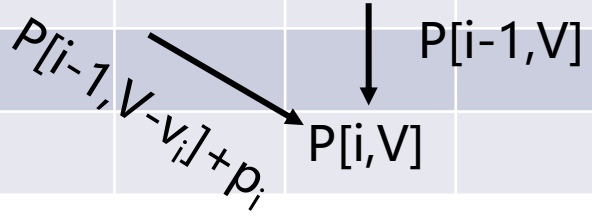
P[i,V]	V=0	1	2	3	…	13	14	15	16
i=0	0	0	0	0	…	0	0	0	0
1	0								
…	0					P[i-1,V]			
N	0				P[i-1,V-v_i]+p_i	P[i,V]			

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

- **确定计算顺序**
  - P[i-1,V-v_i]和P[i-1,V]位于P[i,V]的左上方

P[i,V]	V=0	1	2	3	…	13	14	15	16
i=0	0	0	0	0	…	0	0	0	0
1	0								
…	0								
N	0								



问题：观察子问题依赖关系，如何确定计算顺序?

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

- **确定计算顺序**
  - 按从左到右，从上到下的顺序计算

P[i,V]	V=0	1	2	3	…	13	14	15	16
i=0	0	0	0	0	…	0	0	0	0
1	0								
…	0								
N	0								★

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

实例

v_i	11	4	5	6	5
p_i	22	3	8	11	8

V=16

P[i,V]	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0																
2	0											$v_i>V$					
3	0																
4	0																
5	0																

递推公式：P[i,V] = max {$P[i-1, V-v_i]+p_i$, P[i-1,V] }

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

实例

v_i	11	4	5	6	5
p_i	22	3	8	11	8

V=16

P[i,V]	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0						
2	0																
3	0																
4	0																
5	0																

递推公式：P[i,V] = max {$P[i-1, V-v_i]+p_i$, P[i-1,V] }

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

实例

v_i	11	4	5	6	5
p_i	22	3	8	11	8

V=16

P[i,V]	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0						
2	0																
3	0																
4	0																
5	0																

递推公式：P[i,V] = max {P[0,0]+P_1, P[0,11] }

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

实例

v_i	11	4	5	6	5
p_i	22	3	8	11	8

V=16

P[i,V]	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	22					
2	0																
3	0																
4	0																
5	0																

递推公式：P[i,V] = max {0+22, 0}

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

实例

v_i	11	4	5	6	5
p_i	22	3	8	11	8

V=16

P[i,V]	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	22	22				
2	0																
3	0																
4	0																
5	0																

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

实例

v_i	11	4	5	6	5
p_i	22	3	8	11	8

V=16

P[i,V]	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	22	22	22	22	22	22
2	0	0	0	0	3												
3	0																
4	0																
5	0																

递推公式：P[i,V] = max {0+3, 0}

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

实例

v_i	11	4	5	6	5
p_i	22	3	8	11	8

V=16

P[i,V]	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	22	22	22	22	22	22
2	0	0	0	0	3	3	3	3	3	3	3	22	22	22	22	25	25
3	0	0	0	0	3	8	8	8	8	11	11	22	22	22	22	25	30
4	0	0	0	0	3	8	11	11	11	11	14	22	22	22	22	25	30
5	0	0	0	0	3	8	11	11	11	11	16	22	22	22	22	25	30

最优解

0-1背包问题（动态规划）

伪代码

KnapsackDP(N, p, v, V)

```
1  //求解表格
2  for i ← 1 to N do
3      for c ← 1 to V do
4          if (v[i] ⩽ c) and (p[i] + P[i − 1, c − v[i]] > P[i − 1, c]) then
5              P[i, c] ← p[i] + P[i − 1, c − v[i]]
6          else
7              P[i, c] ← P[i − 1, c]
8          end
9      end
10 end
```

依次计算子问题

时间复杂度O(NV)

问题：如何确定选取了哪些商品？

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

- **递推公式：** $P[i,V] = \max\{P[i-1, V-v_i]+p_i , P[i-1,V]\}$
- **记录决策过程**
  - $Rec[i,V] = \begin{cases} 1, & \text{选择商品} \\ 0, & \text{不选商品} \end{cases}$
- **回溯解决方案**
  - 倒序判断是否选择商品
  - 根据选择结果，确定最优子问题

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

P[i,V]	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	22					
2	0																
3	0																
4	0																
5	0																

Rec	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=1	0	0	0	0	0	0	0	0	0	0	0	1					
2	0												当前最优解				
3	0												包含商品 i				
4	0																
5	0																

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

P[i,V]	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	22	22	22	22	22	22
2	0	0	0	0	3	3	3	3	3	3	3	22					
3	0																
4	0																
5	0																

Rec	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1	0	当前最优解				
3	0												不包含商品 i				
4	0																
5	0																

0-1背包问题（动态规划）

递归计算　P[i, V]: 容量为V时从前i个商品中选的最优解

P[i,V]	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	22	22	22	22	22	22
2	0	0	0	0	3	3	3	3	3	3	3	22	22	22	22	25	25
3	0	0	0	0	3	8	8	8	8	11	11	22	22	22	22	25	30
4	0	0	0	0	3	8	11	11	11	11	14	22	22	22	22	25	30
5	0	0	0	0	3	8	11	11	11	11	16	22	22	22	22	25	30

Rec	V=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i=1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	1	1
3	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	1
4	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

0-1背包问题（动态规划）

伪代码

KnapsackDP(N, p, v, V)

```
1  // 输出最优解方案
2  K ← V
3  for i ← N to 1 do
4      if Rec[i, K] = 1 then
5          print 选择商品 i
6          K ← K − v[i]
7      end
8      else
9          print 不选商品 i
10     end
11 end
12 return P[N, V]
```

0-1背包问题（动态规划）

输出商品-伪代码

KnapsackDP(N, p, v, V)

```
1  // 输出最优解方案
2  K ← V
3  for i ← N to 1 do
4      if Rec[i, K] = 1 then
5          print 选择商品 i
6          K ← K − v[i]
7      end
8      else
9          print 不选商品 i
10     end
11 end
12 return P[N, V]
```

0-1背包递推公式：

$f[i,V] = \max \{ f[i-1,V], f[i-1, V-v_i]+p_i \}$

完全背包?

物品不限制数量

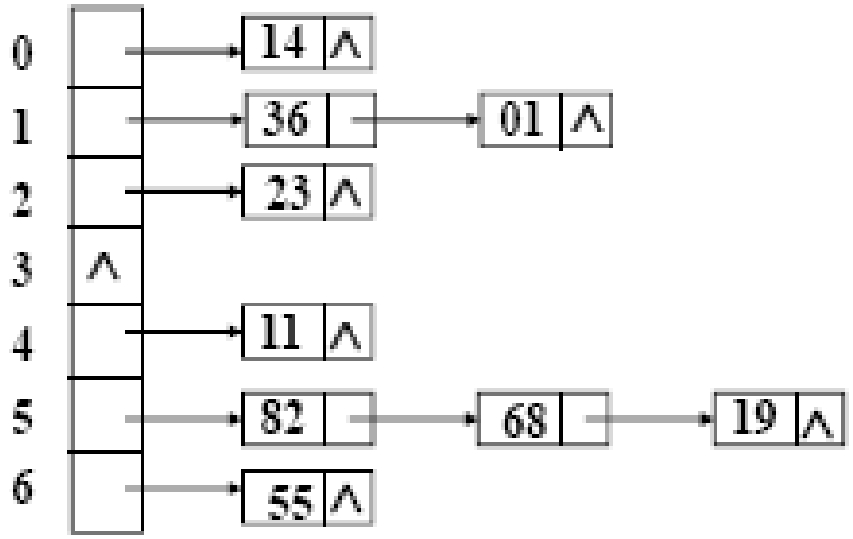
$f[i,V] = \max \{ f[i-1,V], f[i, V-vi]+pi \}$

如何理解

基于散列表的查找

问题

- 步骤
  - Step1: 查找“槽” ： HashKey = H (key)
  - Step2: 在每个槽上挂的线性表上查找
- 每个槽上的线性表是否要求有序?
  - 取决于问题的具体需求
  - 取决于是否有频繁的更新操作
- 每个槽上的线性表如何存储?
  - 链表中的节点包含哪些内容?
- 两种极端情形
  - “槽”很少：极端情况下，退化为单链表
  - “槽”很多：极端情况下，无冲突。即，每个槽要么为空槽，要么槽上只有一个节点。



基于散列表的查找

案例 1

- 问题描述：
  - 统计出给定输入文本中每个单词的重复次数

- 提示
  - HashKey=H(key)
    - 如何选择Key?
    - 可以使用通用Hash函数吗？比如HashPJW, ElfHash
    - Hash表的长度选择
  - 哈希表的存储结构是?

calculated-key =f (key)

Hashkey=calculated-key%Table_Size

基于散列表的查找

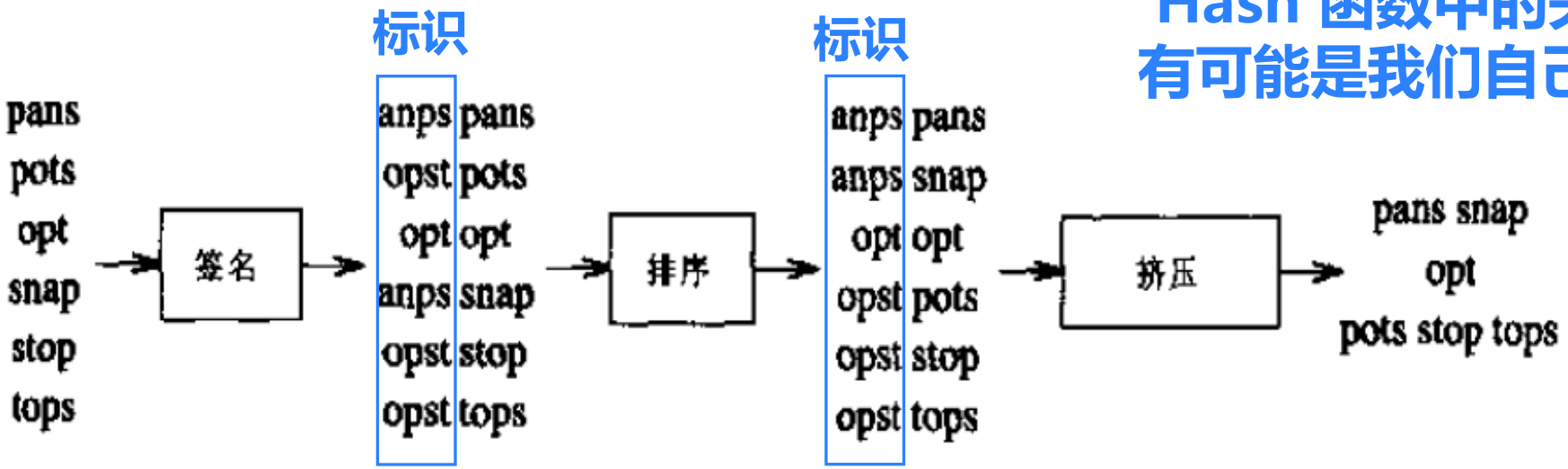
案例 2： 变位词的查找

- 问题描述：
  - 给定一个英语词典。假设有一些时间和空间可以在响应任何查询之前预先处理词典。请查找给定输入单词的所有变位词。例如，“pots”、“stop”和“tops”互为变位词。（《编程珠玑》P15，以及习题2.1）

- 提示
  - 如何选择标识(关键字)?
    - opst:pots,stop,tops
  - 如何集中具有相同标识的单词?
    - 利用hash表来集中具有相同标识的单词（+hash表查找）
    - 按标识排序（标识，单词）元素（+二分查找）

基于散列表的查找

管道处理



Hash 函数中的关键字 Key，
有可能是我们自己构造生成的。

按标识排序+二分查找

（请翻阅《编程珠玑》P18）

基于散列表的查找

挤压

squash

```
int main(void)
{
    char word[WORDMAX],sig[WORDMAX], oldsig[WORDMAX];
     int linenum=0;
     strcpy(oldsig,"");
     while(scanf("%s %s",sig,word)!=EOF){
            if(strcmp(oldsig,sig)!=0 && linenum >0)
                printf("\n");
            strcpy(oldsig,sig);
            linenum++;
            printf("%s ", word);
    }
   printf("\n");
  return 0;
}
```

sign <dict.txt | sort | squash >gramlist

基于散列表的查找

案例 3

- 《编程珠玑》习题1.10：
  - 在成本低廉的隔日送达时代之前，商店允许顾客通过电话订购商品，并在几天后上门自取。商店的数据库使用客户的电话号码作为其检索的关键字（客户知道他们自己的电话号码，而且这些关键字几乎都是唯一的）。你如何组织商店的数据库，以允许高效的插入和检索操作?

- 思考：有内存约束时，无内存约束时，设计的解决方案的不同

- 提示：
  - 利用电话号码的后2位整数作为索引的关键字

基于散列表的查找

案例 4

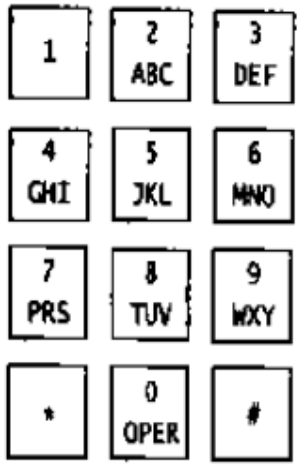
- 《编程珠玑》习题2.6：
  - 20世纪70年代末期，贝尔实验室开发出了“用户操作的电话号码簿辅助程序”，该程序允许雇员使用标准的按键电话在公司电话号码簿中查找电话号码。比如，要查找该系统的设计者 “Mike Lesk”的电话，则按“LESK*M*”（也就是“5375*6*”），随后，系统会输出他的电话号码。这样的服务现在随时可见。但是，该系统中出现的一个问题是，不同的名字可能具有相同的按键编码。在这个Lesk系统中，系统会询问用户更多的信息。

- 思考：编码“5375*6*” 可以代表多少个名字?

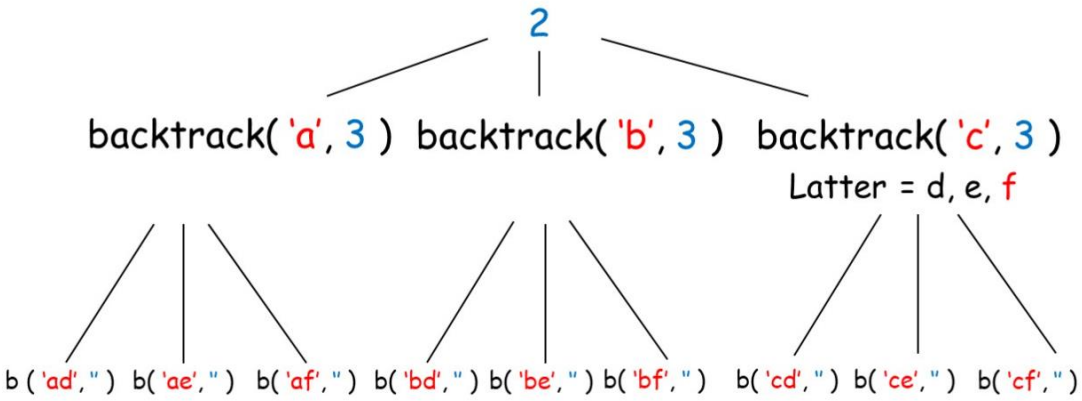
基于散列表的查找

案例 4

- 如何实现一个以按键编码为参数，并返回所有可能的匹配名字的函数?
  - 输入：digits = "23 “
  - 输出：[“AD", “AE", “AF", “BD", “BE", “BF", “CD", “CE", “CF"]




```
phoneMap = {
    "2": "abc",
    "3": "def",
    "4": "ghi",
    "5": "jkl",
    "6": "mno",
    "7": "pqrs",
    "8": "tuv",
    "9": "wxyz",
}
```



基于散列表的查找

案例 4

```
void backtrack(vector<string>& combinations, const
unordered_map<char, string>& phoneMap, const string& digits, int
index, string combination) {
        if (index == digits.length()) {
            combinations.push_back(combination);
        } else {
            char digit = digits[index];
            const string& letters = phoneMap.at(digit);
            for (auto & letter: letters) {
                //combination.push_back(letter);
                backtrack(combinations, phoneMap, digits, index +
1, combination+letter);
                //combination.pop_back();
            }
}}};
```

基于散列表的查找

案例 5：一个实际的搜索问题

- 来自《编程珠玑》13.8 节：如何标识字典
  - 如何用 64KB 的空间存储 30,000 个单词，并能实现较准确的、快速的拼写错误检查。

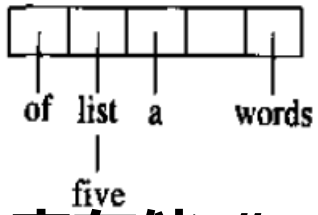
- 注意：是拼写错误检查，而不是纠错。
- 准确性的衡量指标：误判的比率。
  - 误判，是指，存在拼写错误的单词，被判断为正确。

- 思考：若用64KB的空间存储30,000个单词，平均每个单词的长度为多少?

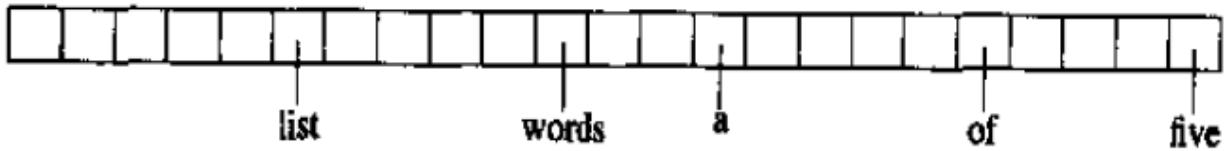
基于散列表的查找

案例 5：一个实际的搜索问题

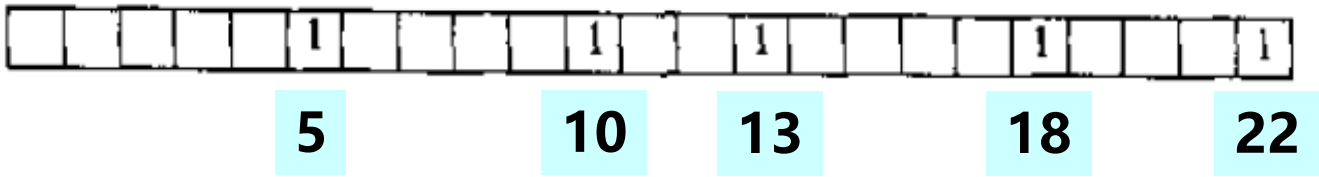
- 利用大小为5的Hash表存储“a list of five words”



- 利用大小为23的Hash表存储“a list of five words”



几乎所有的非空链表上仅包含一个元素!



存储非空槽的槽号，则存储空间为5*ceil(log2 27)=25bit

基于散列表的查找

案例 5：一个实际的搜索问题

- 来自《编程珠玑》13.8 节：如何标识字典
  - 如何用 64KB 的空间存储 30,000 个单词，并能实现较准确的、快速的拼写错误检查。

- 提示：
  - 索引表大小与字典中单词数量差不多，碰撞几率大，需对某个槽上挂的链表进行顺序搜索。
  - 索引表非常大。当索引表大小=227时，几乎所有非空槽上挂的链表仅包含一个元素，且误判概率小（为30000/227≈1/4000）。检索速度快：每个槽中只存放一个bit，通过直接定位槽就可以判断当前单词是否在字典中；但需要227bit，大约16MB空间
  - 只存储字典中每个单词的索引值—需要30000*27bit空间 ≈99KB
  - 进一步利用差值压缩来缩小空间为50KB。

基于散列表的查找

案例 5：一个实际的搜索问题

- 提示：
  - 索引表大小与字典中单词数量差不多，碰撞几率大，需对某个槽上挂的链表进行顺序搜索。
  - 索引表非常大。当索引表大小=2^27时，几乎所有非空槽上挂的链表仅包含一个元素，且误判概率小（为30000/2^27≈1/4000）。检索速度快：每个槽中只存放一个bit，通过直接定位槽就可以判断当前单词是否在字典中；但需要2^27bit，大约16MB空间
  - 只存储字典中每个单词的索引值—需要30000*27bit空间 ≈99KB
  - 进一步利用差值压缩来缩小空间为50KB。
- Questions
  - Q1：Hash表中存储的是什么?
  - Q2：构造Hash表时，可能存在Hash冲突吗?
  - Q3：假设Hash表已构造好，怎么进行拼写错误检查?
  - Q4：用到了BitMap吗?

基于散列表的查找

案例 6

- 问题描述：
  - 给定一个输入的整数，要求从1000万个整数中找出满足以下条件的所有数：这些数模256后的结果值等于输入的指定数模256后的结果值。进一步地，如果原集合中存在输入的指定数，则把这个数从集合中删除，否则把这个数加入集合。

基于散列表的查找

案例 7 ： 箱序列

- 问题描述：
  - 有序输出取值范围为 0-99 的 20 个不重复的随机整数。 （《编程珠玑》P134-135 ）

- 提示：
  - 采用箱数组，每个箱用一个有序链表表示。
  - 映射规则：箱 0 包含 0-24 范围内的整数；箱 1 包含 25-49 范围内的整数；箱 2 包含 50-74 范围内的整数；箱 3 包含 75-99 范围内的整数；
  - 问题定义：将 0-99 之间的整数，放在合适的箱中

基于散列表的查找

案例 7 ： 箱序列

- 问题描述：
  - 有序输出取值范围为 0-99 的 20 个不重复的随机整数。 （《编程珠玑》P134-135 ）

- 步骤：
  - Step1：初始化4个空的有序链表作为箱。
  - Step2：使用某种算法或函数生成20个不重复的随机数
  - Step3：对于每一个生成的随机数，计算其应属于哪个箱，并将其按序插入该箱。
  - Step4：遍历这4个箱，依次输出每个箱中的数字，直到输出了20个数字 （有序）。

基于散列表的查找

案例 7： 箱序列

- 提示：如何实现
  - 数据类型是什么、包含哪写原子操作
- 动态数组+链表
  - 初始化

```
const int BUCKET_NUM = 25;
struct ListNode{
    explicit ListNode(int i=0):mData(i),mNext(NULL){}
    ListNode* mNext;
    int mData;
};

vector<ListNode*> buckets(BUCKET_NUM,(ListNode*)(0));
```

基于散列表的查找

案例 7： 箱序列

- 提示：如何实现
  - 数据类型是什么、包含哪写原子操作
- 动态数组+链表
  - 初始化
  - 插入

```
ListNode* insert(ListNode* head,int val){
    ListNode dummyNode;
    ListNode *newNode = new ListNode(val);
    ListNode *pre,*curr;
    dummyNode.mNext = head;
    pre = &dummyNode;
    curr = head;
    while(NULL!=curr && curr->mData<=val){
        pre = curr;
        curr = curr->mNext;}
    newNode->mNext = curr;
    pre->mNext = newNode;
    return dummyNode.mNext;}
```

基于散列表的查找

案例 7： 箱序列

- 提示：如何实现
  - 数据类型是什么、包含哪写原子操作
- 动态数组+链表
  - 初始化
  - 插入
  - 合并

```
ListNode* Merge(ListNode *head1,ListNode *head2){
    ListNode dummyNode;
    ListNode *dummy = &dummyNode;
    while(NULL!=head1 && NULL!=head2){
        if(head1->mData <= head2->mData){
            dummy->mNext = head1; head1 = head1->mNext;
        }else{
            dummy->mNext = head2;
            head2 = head2->mNext;}
        dummy = dummy->mNext;
    }
    if(NULL!=head1) dummy->mNext = head1;
    if(NULL!=head2) dummy->mNext = head2;
    return dummyNode.mNext;}
```

基于散列表的查找

案例 7： 箱序列

- 提示：如何实现
  - 数据类型是什么、包含哪写原子操作
- 动态数组+链表
  - 初始化
  - 插入
  - 合并

```
void BucketSort(int n,int arr[]){
    vector<ListNode*> buckets(BUCKET_NUM,(ListNode*)(0));
    for(int i=0;i<n;++i){
        int index = arr[i]/BUCKET_NUM;
        ListNode *head = buckets.at(index);
        buckets.at(index) = insert(head,arr[i]);
    }
    ListNode *head = buckets.at(0);
    for(int i=1;i<BUCKET_NUM;++i){
        head = Merge(head,buckets.at(i));
    }
    for(int i=0;i<n;++i){
        arr[i] = head->mData;
        head = head->mNext;}}
```

基于散列表的查找

案例 7+

- 问题描述：
  - 有序输出取值范围为 [0,n] 的 m 个不重复的随机整数。

- 思路?

```
void GenKnuth(int m, int n) {
    for(int i=0; i<n; ++i) {
        if((rand()%(n-i)) < m) {
            cout<<i<<endl;
            --m;
        }
    }
}
```

案例 8：关于重复项的处理

- 问题描述：
  - A,B两个文件，各存放40亿条URL，每条URL占用64字节，内存限制是4G，要求找出A,B文件中共同的URL。

- Q：若直接将40亿条URL读入内存，可以吗？为什么？

- 介绍一种新的数据结构：Bloom Filter

基于散列表的查找

案例 8.1

- 问题描述：
  - A,B两个文件，各存放50亿条URL，每条URL占用64字节，内存限制是4G，要求找出A,B文件中共同的URL。

- 若允许有一定的错误率，可采用 Bloom filter。
- Bloom filter是 BitMap 的扩展
  - 若直接采用 BitMap。那么：
  - 可用内存 4GB=4*1024*1024*1024*8b≈340亿bits。由于每条URL占用64字节，即每个URL对应的整数可以用256^{64} bits >>>340亿bits来描述。因而不能用BitMap。
  - Bloom filter：通过极少的错误换取了存储空间的极大节省。将每条url使用Bloom filter映射为这340亿bit中的k个bit，并将这k个bit置为1。

基于散列表的查找

案例 8.1

- 问题描述：
  - A,B两个文件，各存放50亿条URL，每条URL占用64字节，内存限制是4G，要求找出A,B文件中共同的URL。

- 将文件 a 中的每条 url 使用 Bloom filter 映射为这 340 亿bit；然后依次读取文件 b 中的每条 url，检查其 Bloom filter 映射的 k 比特是否都为1，如果是，那么该 url 应该是两个文件共同的 url（注意：会有一定的错误率）。

基于散列表的查找

Bloom filter

- 是 BitMap 的扩展，它可以将一个数据映射为 BitSet 中的 k bit。通过判断某数对应的 k 个 bit 是否全为 1 可以判断该数是否已存在。
- 适用范围：可以用来实现数据字典，进行数据的判重，或者集合求交集
- 基本原理及要点：位数组+k 个独立 hash 函数。
- 常用操作：
  - (1)将所有的字符串都“记录”到BitSet中（插入）。对各个字符串，利用其计算得到的k个hash值的对应bit置1。
  - (2)检查字符串str是否被BitSet。
  - (3)删除字符串str。

基于散列表的查找

Bloom filter

- 如何将字符串str“记录”到BitSet中?
  - （1）创建一个 m 位 BitSet，先将所有位初始化为 0，然后选择 k 个不同的哈希函数。第 i 个哈希函数对字符串 str 哈希的结果记为 h(i, str)，且 h(i, str) 的范围是 0 到 m-1。
  - （2）对于字符串str，分别计算 h(1, str)，h(2, str) … h(k, str)。然后将 BitSet 的第 h(1, str)、h(2, str)…… h(k, str)位设为 1，从而将字符串 str 映射到 BitSet 中的k bit

基于散列表的查找

Bloom filter

- 如何将字符串str“记录”到BitSet中?

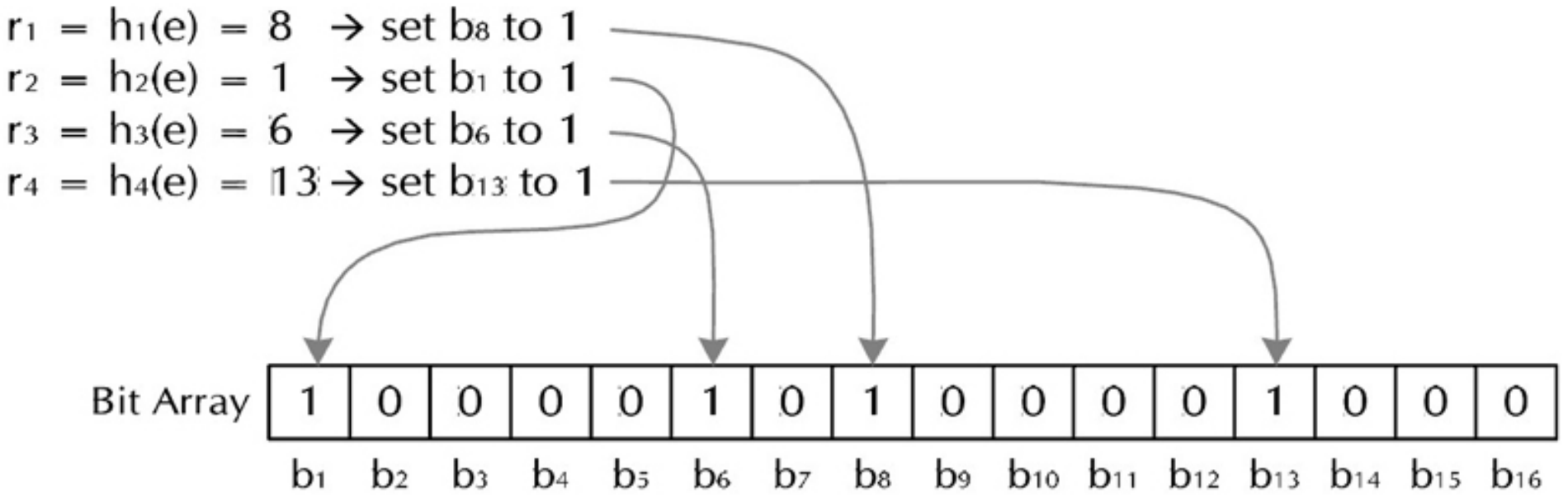


图1.Bloom Filter加入字符串过程

基于散列表的查找

Bloom filter

- 检查字符串 str 是否被 BitSet
  - 利用其计算得到的k个 hash 值的对应 bit 是否全为1。
  - 若全为1，则 “认为”字符串 str 存在；若一个字符串对应的Bit不全为1，则可以肯定该字符串一定没有被 Bloom Filter 记录过。（这是显然的，因为字符串被记录过，其对应的二进制位肯定全部被设为1了）
  - 但是若一个字符串对应的 Bit 全为1，实际上是不能100%的肯定该字符串被 Bloom Filter记录过的。（因为有可能该字符串的所有位都刚好是被其他字符串所对应）这种将该字符串划分错的情况，称为 false positive 。

Bloom filter

- 删除字符串过程
  - 字符串加入了就被不能删除了，因为删除会影响到其他字符串。
  - 可以采用基本Bloom Filter的变体——Counting bloomfilter(CBF)，可以满足删除的要求。
  - CBF将基本Bloom Filter每一个“Bit”改为一个计数器，这样就可以实现删除字符串的功能了。

总结：Bloom filter

- Bloom Filter中冲突概率取决于：哈希函数的个数k，以及Bitset的大小。
- 与Bit-Map的不同之处在于：
  - Bloom Filter使用了k个哈希函数，每个字符串与k个bit对应，从而降低了冲突的概率。K越大，冲突的概率越小。
  - 存在查询结果的**误判**。（但节省了存储开销）
  - 无需处理碰撞，因此它在增加或查找集合元素时所用的时间完全恒定（等于哈希函数的计算时间），无论集合元素本身有多大，也无论多少集合元素已经加入到了位数组中。
- 当要存储的元素个数n很大时，使用bloom filter可以**节省内存开销**。

Bloom filter 分析

- 记 bloom filter 为 m 位，采用 k 个哈希函数，插入数据量为 n
- 假设哈希函数是均匀的，
- 即对于任意一个 bit，在插入一个数据时，其被一个哈希函数置 1 的概率

$$P1 = \frac{1}{m}$$

- 则插入一个数据时，k 个哈希函数均没有将 bit_i 置 1 的概率

$$P0_i = \left(1 - \frac{1}{m}\right)^k$$

Bloom filter 分析

- 考虑 n 个数据插入后，bit_i 仍然为 0，即 n 次插入均未将其置 1，概率：

$$P0_i^{(n)} = (P0_i)^n = \left(1 - \frac{1}{m}\right)^{kn}$$

- 则可知插入完成后 bit_i 为 1 的概率 $P1_i^{(n)} = 1 - P0_i^{(n)} = 1 - \left(1 - \frac{1}{m}\right)^{kn}$
- 此时考虑查询误判率，即该查询的 k 个 bit 均为 1 的概率

$$P(FP) = \prod_{i=i_1,i_2,\ldots i_k} P1_i^{(n)} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

Bloom filter 分析

- 误判率 $P(FP)=\prod_{i=i_1,i_2,\ldots i_k} P1_i^{(n)}=\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^k$
- 这里显然 m 增大时，P(FP) 减小；n 减小时， P(FP) 也减小
- 那么，k 变化时 P(FP) 如何变化?
- 2 个例子：

m = 20, n = 2 时

k = 1： $P(FP)=\left(1-\left(1-\frac{1}{20}\right)^2\right)=0.097$

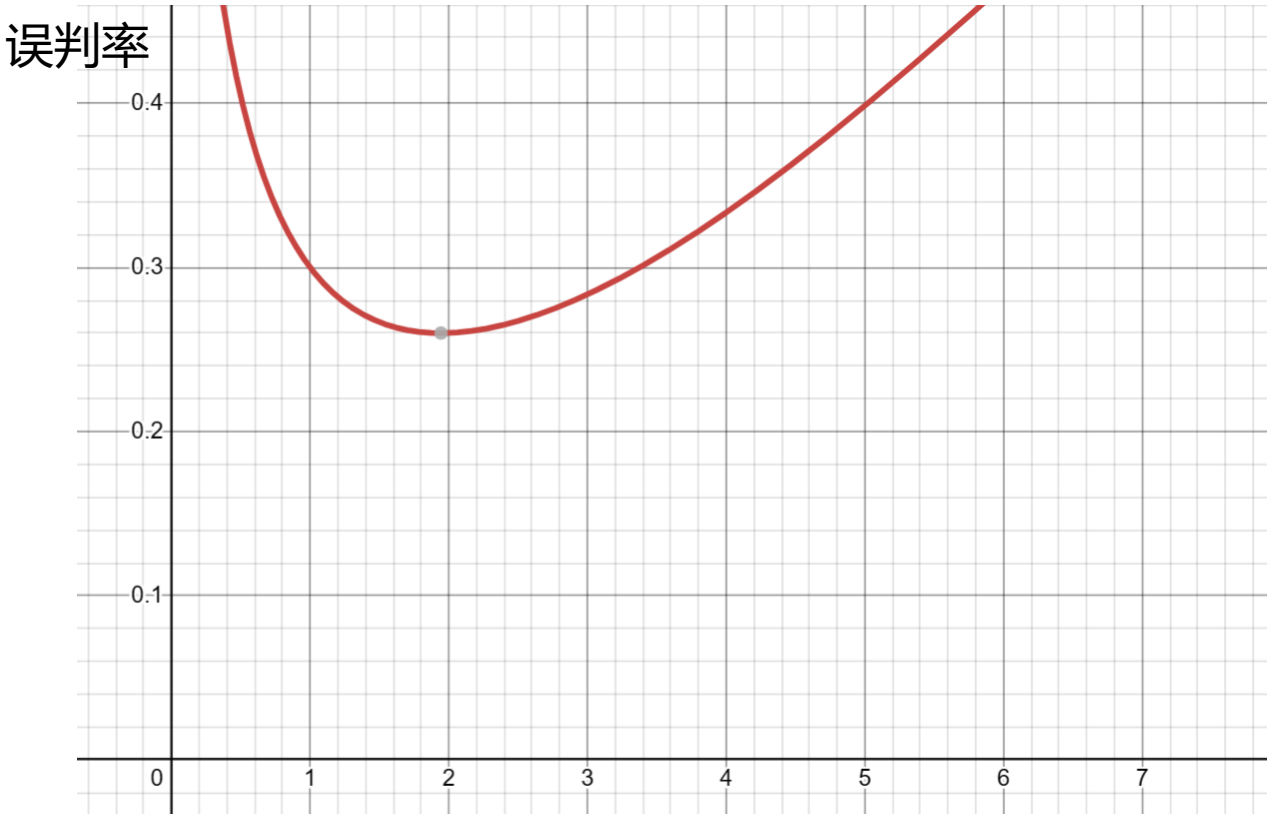
k = 2： $P(FP)=\left(1-\left(1-\frac{1}{20}\right)^4\right)^2=0.034$

令 $a=\left(1-\frac{1}{m}\right)^n\in(0,1)$， $y=f(k)=\left(1-a^k\right)^k, k\geq 1$ ，

观察该函数的图像

Bloom filter 分析

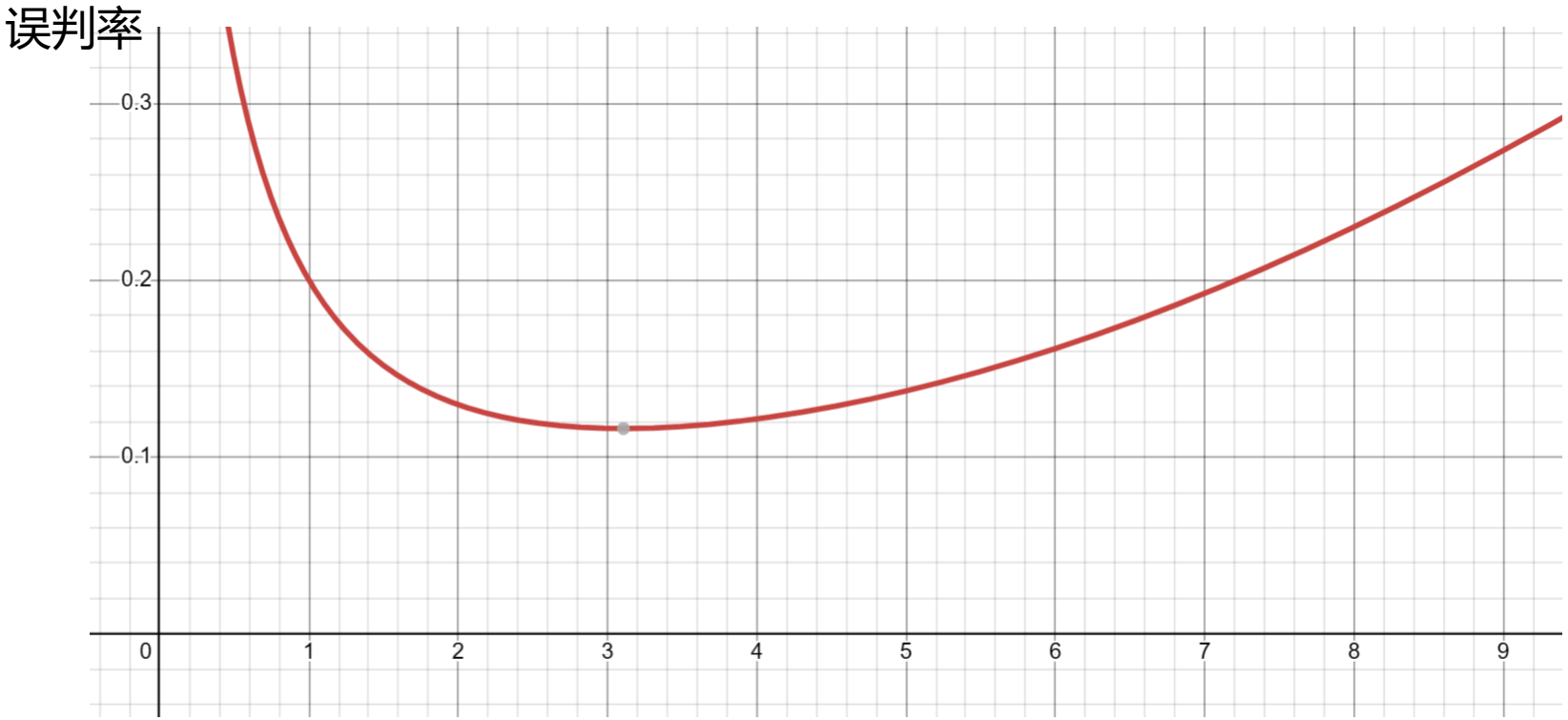
- a = 0.70



基于散列表的查找

Bloom filter 分析

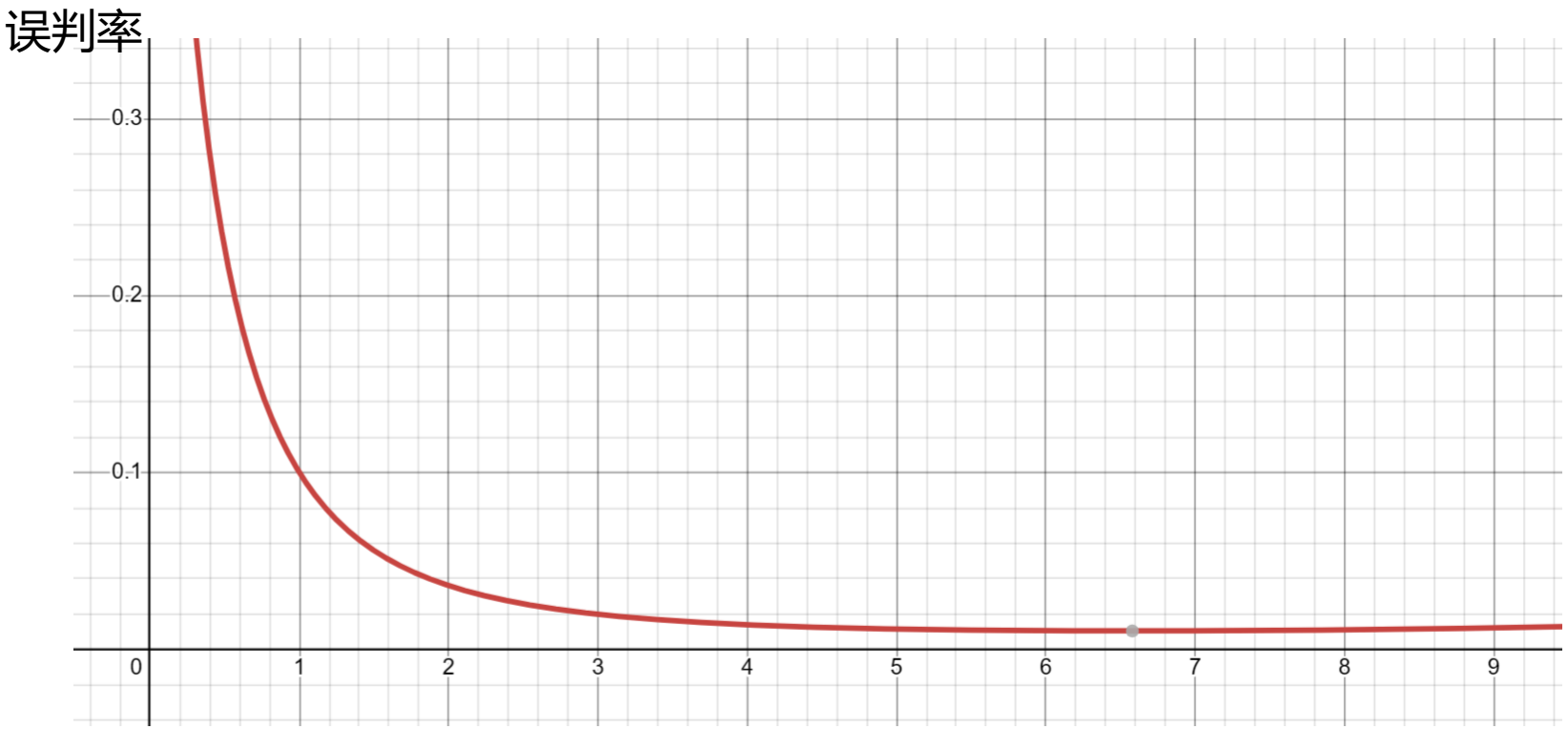
- a = 0.80



基于散列表的查找

Bloom filter 分析

- a = 0.90



Bloom filter 分析

- 可以看出
  - 随着 k 增大，y 会先降低再增大
  - a 越大，y下降的区间越大 —— a 即反应了 m 和 n 之间的关系

- 给定可以接受的错误率以及待插入元素个数，如何求解最小值?

$$P(FP) = \prod_{i=i_1,i_2,\ldots i_k} P1_i^{(n)} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

Practice?

Bloom filter 分析

- 如何根据输入元素个数 n，确定位数组 m 的大小及 hash 函数个数?
  - 当 hash 函数个数 k=(ln2)*(m/n) 时错误率最小。
  - 在错误率不大于 P 的情况下，则 $m \geq n\log_2(1/P)*\log_2 e \approx n\log2\ (1/P)\ *1.44$。
  - 例如：假设要求错误率不超过 0.001，则此时 m 应大概是 n 的14倍，因而 k≈9。

- 注意：这里 m 与 n 的单位不同，m 是 bit 为单位，而 n 则是以元素个数为单位(准确的说是不同元素的个数)。通常单个元素的长度都是有很多 bit 的。所以使用 bloom filter 内存上通常都是节省的。

Bloom filter 分析

- 缺点：
  - 与BitMap类似，通过Bitset无法恢复所表示的数据集合（因为HASH函数是不可逆操作）
  - 不能删除元素。但可以采用其变体Counting Bloom Filter来改善，CBF通过将每一位扩展为一个Counter来解决这一问题。（CBF以牺牲存储性能为代价来改善该问题）

基于散列表的查找

思考：案例 9

- 问题描述：
  - 2.5亿个整数中找出不重复的整数。注意，现有内存空间不足以容纳这2.5亿个整数。

蛮力查找（顺序查找）

目录

- 蛮力查找算法的基本设计思想（教材4.2节P72-73的程序清单4-1 ）
- 蛮力查找实现的优化策略（ 《编程珠玑》 P86)
- 在字符串查找应用场景中使用蛮力查找，应如何提高查找效率？ （教材4.2节P74-75 ）

蛮力查找（顺序查找）

蛮力查找算法的基本设计思想

- 蛮力查找算法的基本设计思想
  - 输入：线性表MyArray，要查找的目标项目SearTarg
  - 输出：若查找失败，则输出-1；若查找成功，则输出匹配项的位置

- 算法设计思路：
  - Step1: 从线性表MyArray中的第一个元素开始，与目标项进行比较，若匹配，则输出匹配项的位置；否则移动到下一个元素继续比较，直至找到匹配项。
  - Step2: 若线性表中所有元素都已比较到，但是仍未找到匹配项。则认为查找失败，输出-1。

蛮力查找（顺序查找）

蛮力查找算法的基本设计思想

- 顺序表上的蛮力查找算法的伪代码示例：

```
int BruteSearch(List MyArray, ElemType SearTarg)
{
        for i=[0,n)
                if (MyArray[i] == SearTarg)
                        return i;
        return -1;
}
```

蛮力查找（顺序查找）

蛮力查找算法的优化（1）

- 伪代码示例：

```
int BruteSearch1(List MyArray, ElemType SearTarg)
{
        hold= MyArray[n];
        MyArray[n]= SearTarg;
        for (i=0; ;i++)
                if (MyArray[i] == SearTarg)
                        break;
         MyArray[n]= hold;
         if i==n
                return -1;
         else
                return i;
}
```

- 优化点：在数组末尾放置一个哨兵值
- 好处：
  - 在循环过程中无需检测是否已到数组末尾。
  - 大约加速了5%
- 最内层循环只包含一次自增、一次数组访问以及一次匹配判断。还可以进一步优化吗?

蛮力查找算法的优化（2）

- 伪代码示例：

```
int BruteSearch1(List MyArray, ElemType SearTarg)
{
        MyArray[n]= SearTarg;
        for (i=0; ;i+=8)
                if (MyArray[i] == SearTarg) {  break; }
                if (MyArray[i+1] == SearTarg) { i + =1; break; }
                if (MyArray[i+2] == SearTarg) { i + =2; break; }
                if (MyArray[i+3] == SearTarg) { i + =3; break; }
                if (MyArray[i+4] == SearTarg) { i + =4; break; }
                if (MyArray[i+5] == SearTarg) { i + =5; break; }
                if (MyArray[i+6] == SearTarg) { i + =6; break; }
                if (MyArray[i+7] == SearTarg) { i + =7; break; }
         if i==n
                return -1;
         else
                return i;
        }
```

- 优化点：将循环展开8次来并行执行（每8次的探查中，才自增1次，自增量取决于元素的匹配判断结果）

- 好处：
  - 将循环展开有助于增加指令的并行性。

蛮力查找（顺序查找）

字符串查找场景中的蛮力查找优化

- 字符串查找场景：
  - 例如，在文本“fffffab cfe defe”中查找字符串“ff”
  - 本质：在字符线性表中查找匹配的子表。
- 在字符串查找应用场景中使用蛮力查找，应如何提高查找效率？（教材4.2节P74-75）
- 程序清单4-2 vs. 程序清单4-1

蛮力查找（顺序查找）

程序清单4-1

```
char * BruteSearch ( const char *text, const char *string )
{
    int len = strlen ( string );

    for ( ; *text; text++ )
        if ( strncmp ( text, string, len ) == 0 )
            return ( (char *) text );

    return ( NULL );
}
```

简单而缓慢！

蛮力查找（顺序查找）

程序清单4-1的改进思路

- 方案1：
  - 仅当字符串text的第一个字符发生匹配时，才需要调用strncmp()函数；
  - 但是，仍需对字符串text的每个字符至少执行两次比较：判断字符是否匹配；判断是否到文本的结尾处
- 方案2：
  - 使用switch语句，实现一个跳转表，那么只需比较一次即可。
  - 但是，C语言中的字符串不允许使用case进行评估。

```
switch ( *text )

{
    case '\0':              /* test for end of text */
            return NULL;
    case *string:           /* test for match with string */
            // call to strncmp()
    default:
            text += 1;  .   /* look at next character */
}
```

能否自己实现一个映射，以实现case评估支持呢?

蛮力查找（顺序查找）

程序清单4-2

- **伪代码示例：**

```
char * BruteSearch ( const char *text, const char *string )
{
    int len  = strlen ( string );

    /* the table. "static" assures its initialized to '\0's */
    static char lookup[UCHAR_MAX + 1];
    lookup[0]                          = 1;     /* EOT process */
    lookup[(unsigned char) (*string)] = 2;     /* a match */

    for ( ;; text++ )
    {   switch ( lookup [(unsigned char) (*text)] )
        {  case 0:
               break;         /* it's not EOT or a match */
           case 1:
               return ( NULL );        /* EOT */
           case 2:
               if ( strncmp
                    ( string + 1, text + 1, len - 1 ) == 0 )
                 return ( (char *) text );     /* a match */
               break;
           default:    /* good coding to include default */
               break;
        }  }
}
```

- **优化点：自定义一个查找表(lookup)，每个表元素表示一个字符。（假设1个字符用8bit表示，则lookup表包含256个元素）**
- **好处：**
  - **每个字符只执行一次比较；**
  - **表lookup查找的速度快；**
  - **查找的准备时间较短（初始化工作在启动时执行，而不是运行时）**
  - **其他：很容易实现不区分大小写的进行字符串查找**
- Q：若string为”school”，那么lookup是怎样的?

字符串哈希

问题描述：

- 给定一个长度为 n 的字符串，再给定 m 个询问，每个询问包含四个整数 l1,r1,l2,r2，请你判断 [l1,r1] 和 [l2,r2] 这两个区间所包含的字符串子串是否完全相同。数据范围$1 \le n,m \le 10^5$

- 输入格式：
  - 第一行: 整数 n和 m，表示字符串长度和询问次数。
  - 第二行: 长度为 n的字符串，字符串中只包含大小写英文字母和数字。
  - 第m行: 每行包含四个整数 l1,r1,l2,r2，表示一次询问所涉及的两个区间。

```
8 3
aabbaabb
1 3 5 7          Yes
1 3 6 8          No
1 2 1 2          Yes
```

字符串哈希

字符串前缀哈希方法：

- 把字符串变成一个P进制数值(哈希值)，实现不同的字符串映射到不同的数字。对形如 $X_1X_2...X_{n-1}X_n$ 的字符串，采用 ASC 码值乘 P 次方计算哈希值。

$$(X_1 \times P^{n-1} + X_2 \times P^{n-2} + \cdots + X_{n-1} \times P^1 + X_n \times P^0) \bmod Q$$

- 注意：
  - 1. 任意字符不可以映射为0, 比如A映射为0, 则A, AA, AAA冲突。
  - 2. 设置P=131或13331, Q=2^64, 一般不会冲突

字符串哈希

- 问题是比较子串是否相同，转化为对应的哈希值是否相同
- 求一个字符串的哈希值就相当于求前缀和，求子串哈希值就相当于求区间和(部分和)。
- 前缀和公式: $h[i+1] = h[i] \times P + s[i], i \in [0, n-1]$，$h$ 为前缀和数组，s为字符串数组
- 区间和公式： $h[l,r] = h[r] - h[l-1] \times P^{r-l+1}$

```
void init(string str)
{
    h[0] = 0;
    p[0] = 1;
    int n = str.size();
    for (int i = 1; i <= n; i++)
    {
        h[i] = h[i - 1] * P + str[i-1];
        p[i] = p[i - 1] * P;
    }}
```

```
ULL query(int l, int r)
{
    return     h[r] - h[l - 1] * p[r-l+1];
}
```