算法概论

第五讲：高级设计与分析技术

薛健

Last Modified: 2019.12.15

主要内容

1 动态规划 **1**

1.1 基本方法 1
1.2 矩阵链乘问题 4
1.3 最优二叉搜索树问题 9
1.4 装配线调度问题 11
1.5 最佳断行问题 14
1.6 动态规划一般步骤 15
1.7 课后习题 16

2 贪心方法 **16**

2.1 基本方法 16
2.2 活动日程安排问题 17
2.3 旅行加油问题 18
2.4 钢管焊接问题 20
2.5 课后习题 22

3 字符串匹配算法 **23**

3.1 基本问题 23
3.2 KMP 算法 23
3.3 BM 算法 26
3.4 近似匹配算法 29

1 动态规划

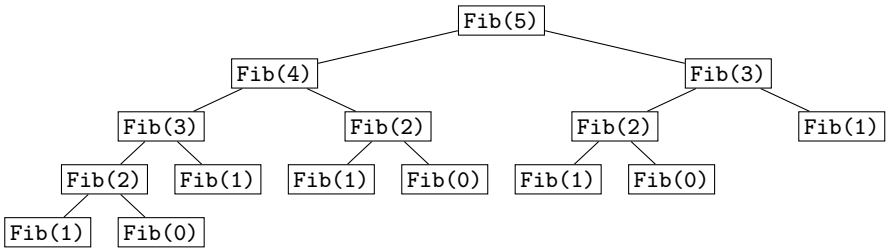
1.1 基本方法

例子

Algorithm `Fib`(n)

```
1 if n < 2 then f ← n;
2 else
3 |   f1 ← Fib(n − 1);
4 |   f2 ← Fib(n − 2);
5 |   f ← f1 + f2;
6 end
7 return f;
```

- 算法时间复杂度达到 $O(c^n)$！
- 在递归调用中，有大量的重复调用，例如在 `Fib(5)` 的计算中，`Fib(2)` 被重复调用了 3 次，而 `Fib(1)` 被调用了 4 次
- 如何改进？



时间复杂度：

$$
\begin{aligned}
W(n) &= W(n-1) + W(n-2) + 1 \\
&\geq 2W(n-2) + 1 \\
&\geq 1 + 2 + 2^2 + \cdots + 2^{n/2} \\
\Rightarrow & W(n) \in \Omega(\sqrt{2}^n) \\
W(n) &\leq 2W(n-1) + 1 \\
&\leq 1 + 2 + 2^2 + \cdots + 2^n \\
\Rightarrow & W(n) \in O(2^n)
\end{aligned}
$$

动态规划

- **动态规划** (Dynamic Programming)是运筹学的一个分支，是求解决策过程 (Decision Process) 最优化的数学方法
- **多阶段决策过程** (Multistep Decision Process)：有一类问题的解决过程可以分为若干个阶段，而在任一阶段后的行为都仅依赖于阶段 i 的过程状态（**最优子结构**），而与阶段 i 如何通过之前的过程达到这种状态的具体方式无关（**无后效性**，即“未来与过去无关”）。

- 20 世纪 50 年代初美国数学家 Richard Bellman 等人在研究多阶段决策过程的优化问题时，根据这类问题多阶段决策的特性，提出了著名的**最优性原理** (Principle of Optimality)，把多阶段过程转化为一系列单阶段问题，逐个求解，从而创立了解决最优化问题的新的算法设计方法——**动态规划**

- 动态规划问世以来，在经济管理、生产调度、工程技术和最优控制等方面得到了广泛的应用。例如最短路线、库存管理、资源分配、设备更新、排序、装载等问题，用动态规划方法比用其它方法求解更为方便和高效

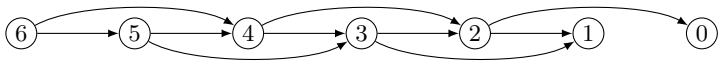
子问题图

- 分治法的基本思路是将原始问题分解成子问题递归求解

Definition 1.1 (子问题图 (Subproblem Graph))**.** 假设算法 A 是解决某个问题的递归算法，A 的子问题图是一个有向图，其结点代表输入数据，从结点 v_i 到 v_j 的有向边 $v_i \to v_j$ 代表在解决输入为 v_i 的子问题时需要递归调用算法 A 解决输入为 v_j 的子子问题。对于算法 A 的某个输入 P，A(P) 的子问题图就是 A 的子问题图中从结点 P 可以到达的部分子图。

- 如果算法 A 总是能够正常终止，则其子问题图必定是**有向无环图** (Directed Acyclic Graph, DAG)，动态规划的本质即是对子问题图中的结点进行逆**拓扑排序** (topological sort)，每一个结点子问题的解决总是依赖于它前一个结点子问题的解决。一个拓扑序列即对应一个解决方案。

- `Fib(6)` 的子问题图



拓扑排序：将偏序集 $(\mathbf{S}, \prec)$ 中的元素排成一行，使得在该序列中元素 a 排在元素 b 的前面当且仅当 $a \prec b$。偏序集一般可用 DAG 表示，则拓扑排序就对应于 DAG 的深度优先搜索。

递归算法的动态规划版本

- 给定递归算法 A，其动态规划版本 (dynamic programming version) $\mathcal{DP}$(A) 是子问题图 A(P) 深度优先遍历过程，且每当处理完一个子问题，将其结果存入字典 soln，由此，递归算法 A 的动态规划版本 $\mathcal{DP}$(A) 可对 A 按如下方式修改得到：

  1. 在递归调用 A(Q) 之前检查 soln 中是否记录了 Q 的结果
     - a 若 soln 中还没有 Q 的结果，则递归调用 A(Q)，即在子问题图中沿 $P \to Q$ 遍历结点 Q
     - b 若 soln 中已有 Q 的结果，则直接取出结果，不再做递归调用
  2. 在返回当前问题 P 的处理结果之前将其存入 soln

- 上述方法仍然采用自顶向下 (top down) 的设计思路
- 暂存子问题结果的方法被称为 memoization
- 一般情况下 $\mathcal{DP}(\mathrm{A})$ 需要一个封装 (wrapper) 调用来初始化存储子问题结果的字典结构

Fib2

- 仍然采用自顶向下 (top-down) 的分治法思想
- 将已经计算过的子问题结果缓存下来 (memoization)

Algorithm Fib2(n)

```
1 map m ← map(0 ← 0, 1 ← 1);
2 return FibRec(m, n);
```

Procedure FibRec(map m, n)

```
1 if m 不包含关键字 n then
2 |   f1 ← FibRec(m, n − 1);
3 |   f2 ← FibRec(m, n − 2);
4 |   m[n] ← f1 + f2;
5 end
6 return m[n];
```

时间复杂度： $O(n)$ **空间复杂度：** $O(n)$

实际上已经不需要递归，可以很容易地转化为循环：

Algorithm Fib2Loop(n)

```
1 f[0] ← 0;
2 f[1] ← 1;
3 for i ← 2 to n do
4 |   f[i] ← f[i − 1] + f[i − 2];
5 end
6 return f[n];
```

Fib3

- 采用自底向上 (bottom-up) 的设计思想，先解小问题，再用小问题的结果解决更大的问题

Algorithm `Fib3`(n)

```
previous ← 0;
current ← 1;
for i ← 1 to n − 1 do
    new ← previous + current;
    previous ← current;
    current ← new;
end
return current;
```

时间复杂度：$O(n)$ 空间复杂度：$O(1)$

1.2 矩阵链乘问题

矩阵链乘法 (Matrix Chain Multiplication)

- 问题描述：
  - 矩阵相乘：需要 rst 次乘法和 $rt(s-1)$ 次加法，我们用乘法次数来衡量矩阵相乘的时间复杂度

$$A_{r\times s}=[a_{ij}],\quad B_{s\times t}=[b_{ij}]$$
$$C_{r\times t}=AB=[c_{ij}]=[\sum_{k=1}^{s}a_{ik}b_{kj}]$$

  - 计算 n 个矩阵的乘积，需要 $n-1$ 次矩阵乘法运算
  - 矩阵乘法满足结合律，因此每次可以任选两个相邻矩阵进行乘法运算
  - **不同的选择次序可以导致不同的总执行时间**
- 目标：**找到一种结合次序使得总的时间消耗最少**

例子

Example 1.1 (4 个矩阵相乘). 有 $A_{30\times1},B_{1\times40},C_{40\times10},D_{10\times25}$ 4 个矩阵，找最快的方式求乘积 $ABCD$

- 总共有 5 种相乘次序：

乘积次序	乘法次数
$(((AB)C)D)$	$30\times1\times40+30\times40\times10+30\times10\times25=20700$
$(A(B(CD)))$	$40\times10\times25+1\times40\times25+30\times1\times25=11750$
$((AB)(CD))$	$30\times1\times40+40\times10\times25+30\times40\times25=41200$
$((A(BC))D)$	$1\times40\times10+30\times1\times10+30\times10\times25=8200$
$(A((BC)D))$	$1\times40\times10+1\times10\times25+30\times1\times25=1400$

- 如果有 n 个矩阵相乘，则可能的结合次序数目将变得十分庞大（等于 Catalan number $C_{n-1} = C_{2n-2}^{n-1}/n$），用穷举法就不合适了

关于 Catalan number：

$$C_n = \frac{C_{2n}^n}{n+1} = \frac{(2n)!}{(n+1)!n!}$$

在组合数学里跟多个计数问题相关，如 C_n 等于给 $n+1$ 个因子加括号的可能性数目（或 $n+1$ 个因子由二元操作符结合次序的可能性数目）；也等于有 $n+1$ 个叶结点的不同满二叉树 (2-tree) 个数。

自顶向下的递归算法设计

- 计算 $A_1A_2\cdots A_n$，其中 A_i 大小为 $d_{i-1} \times d_i$, $1 \le i \le n$
- **回溯算法** (backtracking algorithm)：
  - 选择第一对矩阵乘积的位置 i，求 $B = A_iA_{i+1}$
  - 用 B 代替 A_i 和 A_{i+1}，原问题变成规模为 $n-1$ 的子问题
  - 穷尽第一次相乘位置 i 所有可能的选择，则可求出最优解

Algorithm `MatrixChainOrderTry`$(d[], len, seq[])$

```
1  if len < 3 then bestCost ← 0 ;
2  else
3      bestCost ← ∞;
4      for i ← 1 to len − 1 do
5          c ← 位置 seq[i] 处的乘积开销;
6          newSeq ← seq[0..i − 1, i + 1..len];
7          b ← MatrixChainOrderTry(d, len − 1, newSeq);
8          bestCost ← min(bestCost, b + c);
9      end
10 end
11 return bestCost;
```

$$T(n) = (n-1)T(n-1) + n \Rightarrow T(n) \in \Theta((n-1)!)$$

`MatrixChainOrderTry` 的动态规划版本？

- 子问题图：从原始序列 $0, \cdots, n$ 开始，任意不少于 3 个元素的子序列都可以成为原始问题结点可到达的子问题结点
- 这样的子问题结点数目可达 2^n！
- 动态规划算法设计的一条重要原则：子问题必须可以被简洁地标识（即子问题图的结点数必须得到有效控制）

- 子问题图结点数的阶若能控制在多项式级别，则可以保证所设计出的动态规划算法时间复杂度也能控制在多项式级别
- 对于矩阵链乘问题，我们需要另外的子问题划分方法：找最后一次矩阵相乘的位置
  - $B_{1(d_0\times d_i)} = A_1\cdots A_i$，$B_{2(d_i\times d_n)} = A_{i+1}\cdots A_n$
  - 最后一步是 B_1 和 B_2 相乘，开销为 $d_0d_id_n$
  - 原问题 $(0,n)\Rightarrow$ 子问题 $(0,i)$ 和 (i,n)

新分割方法尝试

Algorithm `MatrixChainOrderTryII`$(d[], lo, hi)$

```
1 if hi − lo = 1 then bestCost ← 0;
2 else bestCost ← ∞;
3 for k ← lo + 1 to hi − 1 do
4 |   a ← MatrixChainOrderTryII(d, lo, k);
5 |   b ← MatrixChainOrderTryII(d, k, hi);
6 |   c ← 位置 k 处的乘积开销;
7 |   bestCost ← min(bestCost, a + b + c);
8 end
9 return bestCost;
```

- 这仍然是一个回溯算法
- 时间复杂度：准确的递归方程比较复杂，但我们可以通过简化后的形式得到它的一个下界：2^n
- 从时间复杂度来看仍然不够好，但我们的目的已经达到

在循环中忽略规模较小的子问题，而只保留规模最大的两个子问题，则可以得到时间复杂度的递归不等式：

$$T(n) \geq 2T(n-1) + n$$

`MatrixChainOrderTryII` 的动态规划版本

- 子问题图：
  - 每个子问题可以用 $0,\cdots,n$ 内的一对整数 (i,j) 表示，$i<j$
  - $0,\cdots,n$ 内可以挑出大约 $n^2/2$ 个这样的整数对
  - 对每一个 (i,j) 以及从 $i+1$ 到 $j-1$ 的 k，有两个子问题需要解决，因此从结点 (i,j) 出发边小于 $2n$
  - 整个子问题图的结点数小于 n^2 而边数小于 n^3
  - 子问题图的深度优先搜索时间复杂度为 $O(n^3)$

- 因此，我们可以写出 MatrixChainOrderTryII 的动态规划版本
- 如何恢复最优链乘开销所对应的链乘次序？
  - 使用另一个字典记录子问题 (i, j) 所对应的最优相乘位置
  - 算出最优解后就可以根据该字典回溯出每个子问题的最后相乘位置，也就可以得到整个矩阵链的相乘次序
- 时间复杂度：与子问题图中的结点数和边数相关，在动态规划算法中，每个结点和每条边只需被处理和访问一次，因此时间复杂度为 $O(n^3)$，比指数阶的复杂度要好得多

MatrixChainOrderTryIIDP

Algorithm MatrixChainOrderTryIIDP($d[], lo, hi, cost[][]$)

```
1  if hi − lo = 1 then bestCost ← 0;
2  else bestCost ← ∞;
3  for k ← lo + 1 to hi − 1 do
4      if Member(cost, lo, k) then a ← Retrieve(cost, lo, k) ;
5      else a ← MatrixChainOrderTryIIDP(d, lo, k, cost) ;
6      if Member(cost, k, hi) then b ← Retrieve(cost, k, hi) ;
7      else b ← MatrixChainOrderTryIIDP(d, k, hi, cost) ;
8      c ← 位置 k 处的乘积开销;
9      if bestCost > a + b + c then
10         bestCost ← a + b + c; bestPos ← k;
11     end
12 end
13 Store(cost, lo, hi, bestCost); Store(last, lo, hi, bestPos);
14 return bestCost;
```

自底向上 (bottom up) 的设计思路

- 如果能够找到一种方便的方式生成子问题图的逆拓扑排序，则可以采用自底向上的方式来设计算法，消除递归：
  1. 对每个矩阵计算最优乘积次序，显然对单个矩阵不需要乘法
  2. 对每两个相邻矩阵计算最优乘积次序，显然只有一种次序
  3. 对每三个相邻矩阵计算最优乘积次序，需要用到上面两步的结果
  4. 假设任意 k 个相邻矩阵的最优乘积次序已经得到，则根据已知结果计算 $k + 1$ 个相邻矩阵的最优乘积次序
  5. 得到任意 n 个相邻矩阵的最优乘积次序，算法结束
- 对于子问题 (i, j)，若固定 i，随 j 增加，对应逆拓扑序列中结点 (i, j) 之后的结点，而固定 j，随 i 减少，也对应逆拓扑序列中结点 (i, j) 之后的结点，因此可以用两重循环来构造逆拓扑序列，外层循环对应 i 递减，内层循环对应 j 递增

算法描述

Algorithm `MatrixChainOrder`$(d[], n)$

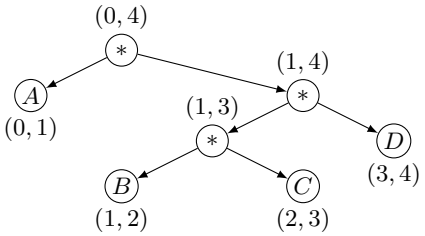
```
1  for lo ← n − 1 down to 0 do
2      for hi ← lo + 1 to n do
3          if hi − lo = 1 then bestCost ← 0; bestPos ← −1;
4          else bestCost ← ∞;
5          for k ← lo + 1 to hi − 1 do
6              c ← cost[lo][k] + cost[k][hi] + d[lo]d[k]d[hi];
7              if c < bestCost then
8                  bestCost ← c;
9                  bestPos ← k;
10             end
11         end
12         cost[lo][hi] ← bestCost; last[lo][hi] ← bestPos;
13     end
14 end
15 return cost and last;
```

时间复杂度：$\Theta(n^3)$

例子

$$A_{30\times 1} \times B_{1\times 40} \times C_{40\times 10} \times D_{10\times 25}$$

$$cost = \begin{bmatrix} \cdot & 0 & 1200 & 700 & 1400 \\ \cdot & \cdot & 0 & 400 & 650 \\ \cdot & \cdot & \cdot & 0 & 10000 \\ \cdot & \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} \quad last = \begin{bmatrix} \cdot & -1 & 1 & 1 & 1 \\ \cdot & \cdot & -1 & 2 & 3 \\ \cdot & \cdot & \cdot & -1 & 3 \\ \cdot & \cdot & \cdot & \cdot & -1 \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$



$\Rightarrow \quad \boldsymbol{A((BC)D)}$

提取链乘次序

Algorithm `ExtractOrder`$(n, last[][])$

```
1  global next ← 0;
2  mult ← Array(n − 1);
3  ExtractOrderRec(0, n, last, mult);
```

Procedure `ExtractOrderRec`$(lo, hi, last[][], mult[])$

```
1 if hi − lo > 1 then
2     k ← last[lo][hi];
3     ExtractOrderRec(lo, k, last, mult);
4     ExtractOrderRec(k, hi, last, mult);
5     mult[next] ← k;
6     next ← next + 1;
7 end
```

时间复杂度：$T(n) = 2n - 1 \in \Theta(n)$　递归深度：$\Theta(n)$

1.3 最优二叉搜索树问题

问题描述

- 回顾：二叉搜索树 (Binary Search Tree)
  - 用中序 (inorder) 遍历二叉搜索树可得到从小到大排序的元素序列
  - 包含相同元素的二叉搜索树平衡度 (degrees of balance) 可以差别很大
  - `BSTSearch`：搜索所需比较次数与所找结点在树中的深度有关，最坏情况搜索时间复杂度为 $\Theta(n)$，如果二叉搜索树尽可能达到平衡，则最坏情况复杂度可降到 $\Theta(\log n)$
- 设可供搜索的元素键值为 $K_1, K_2, \cdots, K_n$，每个元素被搜索的可能性为 $p_1, p_2, \cdots, p_n$
- 将所有元素放入一棵二叉搜索树中，设找到 K_i 所需的键值比较次数为 c_i，则二叉搜索树 T 的平均比较次数为：

$$A(T) = \sum_{i=1}^{n} p_i c_i$$

- 若每个元素被搜索的可能性不相等，则平衡二叉树未必能得到较低的 $A(T)$

问题描述 (cont.)

- **构造最优二叉搜索树：**
  - 给定元素键值 $K_1, K_2, \cdots, K_n$ 及其被搜索的概率 $p_1, p_2, \cdots, p_n$，构造一棵二叉搜索树使搜索的平均比较次数 $A(T)$ 最小
- 基本思路：
  - 假定 $K_1, K_2, \cdots, K_n$ 按从小到大排序

  - 若选择 K_i 为二叉搜索树的根结点，则其左子树包含 $K_1, \cdots, K_{i-1}$，右子树包含 $K_{i+1}, \cdots, K_n$
  - 则问题转化为两个子问题：构造最优左子树和右子树
  - 根结点取遍所有 K_i 即可找到最优解

- 与矩阵链乘问题类似，子问题可用整数对 $(low, high)$ 表示，即构造 $K_{low}, \cdots, K_{high}$ 的最优二叉搜索树

- 每个子问题的时间开销（平均比较次数）计算需要进一步考察

子问题时间开销的计算

- 几个定义：
  - $\boldsymbol{A(low, high, r)}$：选择 K_r 为根结点时子问题 $(low, high)$ 的最小带权搜索开销
  - $\boldsymbol{A(low, high)}$：子问题 $(low, high)$ 的最小带权搜索开销（取遍所有可能 K_r 作为根结点）
  - $\boldsymbol{p(low, high)} = \sum_{i=low}^{high} p_i$：实际上是所搜索的元素键值落在区间 $[K_{low}, K_{high}]$ 中的概率

- 包含 $K_{low}, \cdots, K_{high}$ 的二叉搜索树的带权搜索开销为 W，其根结点深度为 0，若将其作为某一个新的根结点的子树，其根结点深度变为 1，其带权搜索开销则变为 $W + p(low, high)$

- 建立递推方程：

$$\begin{aligned} A(low, high, r) &= p_r + p(low, r-1) + A(low, r-1) \\ &\quad + p(r+1, high) + A(r+1, high) \\ &= p(low, high) + A(low, r-1) + A(r+1, high) \\ A(low, high) &= \min\{A(low, high, r) \,|\, low \le r \le high\} \end{aligned}$$

算法描述

Algorithm `OptimalBST`$(p[], n)$

```
1 for low ← n + 1 down to 1 do
2 |   for high ← low − 1 to n do
  |     BestChoice(p, cost, root, low, high) ;
3 end
4 return cost and root;
```

Procedure BestChoice$(p[], cost[][], root[][], low, high)$

```
1 if high < low then bestCost ← 0; bestRoot ← −1 ;
2 else bestCost ← ∞;
3 for r ← low to high do
4 |   c ← p(low, high) + cost[low][r − 1] + cost[r + 1][high];
5 |   if c < bestCost then bestCost ← c; bestRoot ← r ;
6 end
7 cost[low][high] ← bestCost;
8 root[low][high] ← bestRoot;
```

时间复杂度：$\Theta(n^3)$ 如何计算 $\mathrm{p}(i,j)$ ？

- $\mathrm{p}(i,j)$ 可以通过 $\Theta(n)$ 的预处理得到在算法运行过程中 $\Theta(1)$ 的复杂度
- $\mathrm{p}(i,j)$ 的计算方法：
  - 方法 1：用一个 $\Theta(n^2)$ 的预处理过程计算每一个 $\mathrm{p}(i,j)$ (注意：$i \le j$)，这样在 BestChoice 中可以用 $\Theta(1)$ 的时间取得 $\mathrm{p}(low, high)$
  - 方法 2：保持 BestChoice 中 $\mathrm{p}(low, high)$ 的复杂度为 $\Theta(1)$，但预处理的复杂度可以降到 $\Theta(n)$，考虑下面的关系：

    $$\mathrm{p}(i,j) = \mathrm{p}(1,j) - \mathrm{p}(1,i)$$

    因此在预处理步骤只要计算出 $\mathrm{p}(1,1)$ 到 $\mathrm{p}(1,n)$ 的值即可，显然该步骤时间复杂度为 $\Theta(n)$

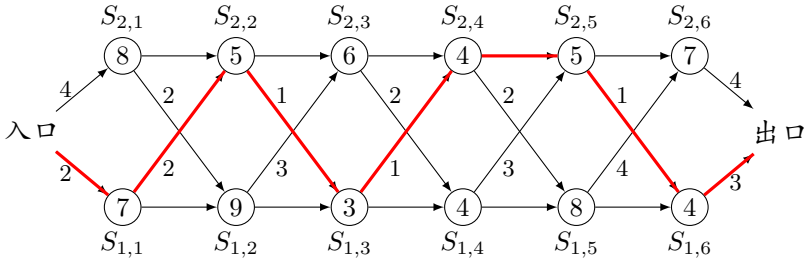
1.4 装配线调度问题

装配线调度问题

- **问题描述**
  - 假定某汽车生产线有两条装配线，待装配的汽车底盘可选择进入任一条装配线，经过 n 道工序进行装配，每道工序负责添加某些零件，当最后一道工序完成即装配成整车
  - 记两条装配线上的工序分别为 $S_{1,1}, S_{1,2}, \cdots, S_{1,n}$ 和 $S_{2,1}, S_{2,2}, \cdots, S_{2,n}$
  - 假设装配线 1 上的工序 $S_{1,k}$ 和装配线 2 上的工序 $S_{2,k}$ $(1 \le k \le n)$ 完成同样的装配工作，因此从 $S_{1,k-1}$ 可以选择进入 $S_{1,k}$ 或 $S_{2,k}$ 继续下一道工序，但若要从装配线 1 切换到装配线 2 需要额外的转换时间 $t_{1,k-1}$，从装配线 2 切换到装配线 1 与之类似
  - 完成每道工序所需时间为 $a_{i,k}$ $(i = 1, 2; 1 \le k \le n)$，进入装配线和移出装配线的时间分别为 e_i 和 x_i $(i = 1, 2)$
- **目标**：找到一种调度方法使得总的装配时间最短

例子

Example 1.2 (有 6 道工序的装配线).



实际上就是求从“入口”到“出口”的带权最短路径。

算法设计

- 问题分割：
  - 子问题 (i,k)：装配线 i 上的第 k 道工序完成时已消耗时间的最小值 $(i=1,2;1\le k\le n)$
- 是否具有最优子结构性质：
  - 假设 $s,s_1,s_2,\cdots,s_n,t$ 是从入口 s 到出口 t 的最优装配路径
  - 其子路径 $s,s_1,s_2,\cdots,s_i$ 一定是从 s 到 s_i 的最优装配路径，否则，若存在另一条从 s 到 s_i 的更快的装配路径 $s,s'_1,\cdots,s'_{i-1},s_i$，则 $s,s'_1,\cdots,s'_{i-1},s_i,s_{i+1},\cdots,t$ 是比 $s,s_1,s_2\cdots,s_n,t$ 更优的装配路径，与原假设矛盾
- 建立子问题求解的递推方程：
  - $f_{i,k}$ 记录装配线 i 上的第 k 道工序完成时已消耗时间的最小值，则：

$$
\begin{aligned}
f_{1,1} &= e_1 + a_{1,1} \\
f_{2,1} &= e_2 + a_{2,1} \\
f_{1,k+1} &= \min\{(f_{1,k} + a_{1,k+1}),\ (f_{2,k} + a_{1,k+1} + t_{2,k})\} \\
f_{2,k+1} &= \min\{(f_{2,k} + a_{2,k+1}),\ (f_{1,k} + a_{2,k+1} + t_{1,k})\}
\end{aligned}
$$

算法描述

Algorithm `FastestWay`$(f[][], a[][], t[][], n, e1, e2, x1, x2)$

```
1  f[1][1] ← e1 + a[1][1];
2  f[2][1] ← e2 + a[2][1];
3  l ← array[2][n];
4  for i ← 2 to n do
5  |   StepForward(f, a, t, l, i);
6  end
7  if f[1][n] + x1 ≤ f[2][n] + x2 then
8  |   totalf ← f[1][n] + x1;
9  |   lastl ← 1;
10 else
11 |   totalf ← f[2][n] + x2;
12 |   lastl ← 2;
13 end
14 PrintSchedure(l, n, lastl);
```

`StepForward`

Procedure `StepForward`$(f[][], a[][], t[][], l[][], i)$

```
1  if f1[i − 1] + a1[i] ≤ f2[i − 1] + a1[i] + t2[i − 1] then
2  |   f1[i] ← f1[i − 1] + a1[i];
3  |   l[1][i] ← 1;
4  else
5  |   f1[i] ← f2[i − 1] + a1[i] + t2[i − 1];
6  |   l[1][i] ← 2;
7  end
8  if f2[i − 1] + a2[i] ≤ f1[i − 1] + a2[i] + t1[i − 1] then
9  |   f2[i] ← f2[i − 1] + a2[i];
10 |   l[2][i] ← 2;
11 else
12 |   f2[i] ← f1[i − 1] + a2[i] + t1[i − 1];
13 |   l[2][i] ← 1;
14 end
```

`PrintSchedure`

Procedure PrintSchedure($l[][], n, lastl$)

```
1 i ← lastl;
2 k ← n;
3 repeat
4 |   print “装配线” i, “工序” k − 1;
5 |   i ← l[i][k];
6 |   k ← k − 1;
7 until k < 2;
```

1.5 最佳断行问题

问题描述

- 在文字排版技术中存在这样的问题：
  - 如何给一个英文单词序列断行，使构成的段落排版结果最美观？
- **最优断行** (Optimization Line-breaking) 问题：
  - 在 Knuth 为 “The Art of Computer Programming” 设计的排版系统 TEX 中引入
  - 输入：n 个单词的长度 $w_1, \cdots, w_n$ 及排版行宽 W
  - 输出：在行宽为 W 的限制下，n 个单词构成的段落
- 问题的简化：
  - 假定字母的宽度相等，且每个单词后面跟一个空格，则 w_i 可由构成单词的字母个数加 1 来表示
  - W 以每一行可放下的字母个数表示，则从第 i 个单词到第 j 个单词若要排在同一行中必须满足：$\sum_{k=i}^{j} w_k \leq W$，而这一行的剩余空白长度为 $X = W - \sum_{k=i}^{j} w_k$
  - 剩余的空白会影响美观，因此可以构造一个惩罚函数来反映这种情况，例如取 X^3，但段落的最后一行例外
  - 目标：使每一行惩罚函数值之和最小

算法设计思路

- 子问题分割：
  - 将原问题 $(1, n)$ 分割为 $(1, k)$ 和 $(k + 1, n)$
  - k 有可能是一个很糟糕的断行位置
  - 段落最后一行不计惩罚，但不包含段落最后一行的所有子问题各自的最后一行照常需要计惩罚
  - 有两类不同结构的子问题

  - 需要证明：取遍所有 k，$(1,k)$ 和 $(k+1,n)$ 相结合的最优结果就是原问题 $(1,n)$ 的最优结果

- 有没有更好的分割方案？
  - 在上面的分析中有一类子问题只需要一个整数来表示：(i,n)
  - 另一类子问题是否还需要？$\Rightarrow$ Method99
  - 取遍所有的 k 使得单词 1 到 k 可以放进第一行，剩下的子问题就是如何对单词 $k+1$ 到 n 在段落剩下的部分中断行（从第二行开始）
  - 综合第一行的惩罚函数值和子问题的惩罚函数值，其中和最小的一个即对应原始问题的最优解

算法描述

Algorithm `LineBreak`$(w[], W, i, n, L)$

```
1  if ∑_{k=i}^{n} w[k] ≤ W then
2      单词 i 到 n 放入第 L 行 ; penalty ← 0;
3  else
4      k ← 1; penalty ← ∞; kmin ← 0;
5      while ∑_{t=i}^{i+k-1} w[t] ≤ W do
6          X ← W − ∑_{t=i}^{i+k-1} w[t];
7          kp ← LineCost(X)+ LineBreak(w, W, i + k, n, L + 1);
8          if kp < penalty then penalty ← kp; kmin ← k ;
9          k ← k + 1;
10     end
11     将单词 i 到 i + kmin − 1 放入第 L 行;
12 end
13 return penalty;
```

`LineBreak` $\Rightarrow$ `LineBreakDP`　时间复杂度：$\Theta(Wn) = \Theta(n)$

对每一行的 k，至多有 $W/2$ 种选择，子问题图中约有 n 个结点，每个结点至多发出 $W/2$ 条边，W 通常可看作常数，因此 `LineBreakDP` 的时间复杂度为 $\Theta(Wn) = \Theta(n)$。

1.6 动态规划一般步骤

动态规划算法设计的一般步骤

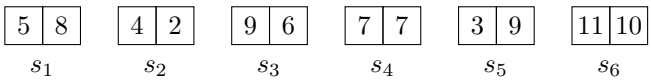
1. 使用自顶向下 (top down) 的方式分析问题，给出一般递归算法（复杂度可能是指数级的）
2. 分析所得递归算法的子问题图，看是否可以用暂存子问题结果 (memoization) 的方式减少子问题的重复计算，若是，则可用固定的转换步骤将一般递归算法转换为其动态规划版本，此处的关键是：子问题的标识应尽可能简洁，以便限制子问题图的规模和所需暂存空间的大小

3. 根据子问题图（深度优先搜索）分析动态规划版本的递归算法时间复杂度，看是否达到要求
4. 设计用于暂存子问题结果的字典数据结构
5. 分析子问题图的结构，看是否可以简单地得到结点的逆拓扑序列，若是则可以按自底向上 (bottom up) 的方式设计出动态规划算法的非递归版本
6. 确定从字典数据中提取最优解（决策过程）的方法

1.7 课后习题

“多米诺骨牌”问题的动态规划算法

Exercise (6). 现有 n 块“多米诺骨牌” $s_1, s_2, \cdots, s_n$ 水平放成一排，每块骨牌 s_i 包含左右两个部分，每个部分赋予一个非负整数值，如下图所示为包含 6 块骨牌的序列。骨牌可做 180 度旋转，使得原来在左边的值变到右边，而原来在右边的值移到左边，假设不论 s_i 如何旋转，$L[i]$ 总是存储 s_i 左边的值，$R[i]$ 总是存储 s_i 右边的值，$W[i]$ 用于存储 s_i 的状态：当 $L[i] \leq R[i]$ 时记为 0，否则记为 1，试设计时间复杂度为 $O(n)$ 的动态规划算法求 $\sum_{i=1}^{n-1} R[i] \cdot L[i+1]$ 的最大值，以及当取得最大值时每个骨牌的状态。下面是 $n = 6$ 时的一个例子。



deadline: 2019.12.29

2 贪心方法

2.1 基本方法

贪心方法

- 与动态规划方法一样，贪心方法一般也用来解决某种优化问题，与动态规划不同的是贪心方法并不穷尽所有子问题的最优解
- 基本思路：
  - 在一个决策序列中，每一步单独的决策其优劣有一个度量标准来衡量
  - 每一步决策总是选择在度量标准下最优的那个分支（决定）
  - 每一步决策一旦决定便不再更改（不同于回溯算法）
- 几点注意：
  - 贪心方法给出的解不一定是（全局）最优解

  - 给定具体问题设计贪心算法，一般需要证明所设计的算法求得的确实是给定问题的最优解
  - 对同一个问题，贪心算法的效率一般要高于动态规划算法

- 经典贪心算法实例：
  - Prim 算法、Dijkstra 算法、Kruskal 算法
  - 将在图算法一讲具体介绍

2.2 活动日程安排问题

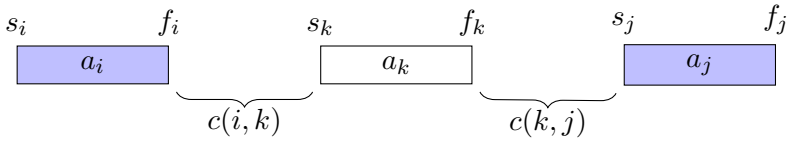
问题描述

- 某一公共场地计划安排 n 个不同活动：$a_1, a_2, \cdots, a_n$
- 活动 a_i 的开始时间和结束时间分别为 s_i 和 f_i $(0 \le s_i < f_i < \infty)$，因此，如果 a_i 得以进行，则公共场地将在 $[s_i, f_i)$ 时间段内被占用
- 两个不同活动 a_i 和 a_j 在时间上不存在冲突，则称这两个活动是相容的，即 $f_i \le s_j$ 或 $f_j \le s_i$
- **活动日程安排**：从活动 $a_1, a_2, \cdots, a_n$ 中选择时间不冲突的活动在公共场地举行
- **优化目标**：安排的活动尽可能多，即从 $\mathbf{S} = \{a_1, a_2, \cdots, a_n\}$ 中选出一个相容子集并使其中包含的元素最多（场地的利用率最高）

动态规划方法

- 子问题：$\mathbf{S}(i,j) = \{a_k \in \mathbf{S} | f_i \le s_k < f_k \le s_j\}$
- 令 $f_0 = 0,\ s_{n+1} = \infty$，则原问题可用 $\mathbf{S}(0, n+1)$ 表示
- 将所有活动按结束时间从小到大排序，则当 $i \ge j$ 时 $\mathbf{S}(i,j) = \varnothing$
- 若用 $c(i,j)$ 表示子问题 $\mathbf{S}(i,j)$ 最优解所安排的活动个数，则：

$$c(i,j) = \begin{cases} 0 & \mathbf{S}(i,j) = \varnothing, \\ \max\limits_{i<k<j} \{c(i,k) + c(k,j) + 1\} & \mathbf{S}(i,j) \ne \varnothing. \end{cases}$$



- 写出动态规划算法（课后练习）

贪心算法设计

- 贪心方法思路：每次取剩下活动中结束时间最早且与已安排活动不冲突的的那个加入日程安排

Algorithm `GreedyActivitySelector`$(a[], n)$

```
1  按 a[i].f 从小到大对 a 排序;
2  A ← {a[1]};
3  i ← 1;
4  for m ← 2 to n do
5      if a[m].s ≥ a[i].f then
6          A ← A ∪ {a[m]};
7          i ← m;
8      end
9  end
10 return A;
```

时间复杂度：$\Theta(n \log n)$

算法的正确性

Theorem 2.1. 令 $a_m \in \mathbf{S}(i,j)$ 且 $f_m = \min\{f_k | a_k \in \mathbf{S}(i,j)\}$，则有：

1. a_m 在 $\mathbf{S}(i,j)$ 的最大相容子集中；
2. $\mathbf{S}(i,m) = \varnothing$

Proof. 先看第 2 个结论，若假设 $\mathbf{S}(i,m) \neq \varnothing$，则存在 a_k 使得 $f_i \leq s_k < f_k \leq s_m < f_m$，这与 f_m 在 $\mathbf{S}(i,j)$ 中最小矛盾，因此 $\mathbf{S}(i,m) = \varnothing$；再看第 1 个结论，设 $\mathbf{A}(i,j)$ 是 $\mathbf{S}(i,j)$ 的最大相容子集，若假设 $a_m \notin \mathbf{A}(i,j)$，对 $\mathbf{A}(i,j)$ 中的元素按结束时间从小到大排序，设 a_k 是排序后第一个元素，则对 $\mathbf{A}(i,j)$ 中任意其他元素 a_x 均有 $f_k \leq s_x$，现构造 $\mathbf{A}'(i,j) = (\mathbf{A}(i,j) - \{a_k\}) \cup \{a_m\}$，则有 $f_m \leq f_k \leq s_x$，因此 $\mathbf{A}'(i,j)$ 也是一个相容子集，且元素数目与 $\mathbf{A}(i,j)$ 相同，所以 $\mathbf{A}'(i,j)$ 也是最大相容子集（另一个最优解）。 □

2.3 旅行加油问题

问题描述

- 开车从城市 A 到城市 B，途经 n 个加油站
- 出发点和终点的加油站标记为第 0 个和第 $n+1$ 个加油站
- 每个加油站与前一个加油站之间的距离为 d_i $(d_0 = 0)$
- 每个加油站油价为 p_i (已折算成每公里耗油的油价，$p_{n+1} = 0$)

- 汽车加满油可行驶 L 公里
- 在城市 A、B 之间的任意 L 公里路程上至少有 1 个加油站，至多有 k 个加油站
- 出发前油箱是空的
- **问题**：在每个加油站应该加多少油？（折算成能行驶的公里数）
- **优化目标**：油费最少

贪心算法设计

- 假设当行至加油站 i 时，油箱剩余的油还能走 r_i 公里
- 在加油站 i 需加 g_i 公里油
- 则要确定 g_i 和下一个停车加油的加油站，需分两种情况讨论：

1. 若在 L 公里内，u 是第一个油价低于 i ($p_u \leq p_i$) 的加油站
    - 加油量 $g_i = \sum_{j=i+1}^{u} d_j - r_i$，即刚够行驶到 u
    - 任何最优方案在从加油站 i 到加油站 u 的过程中至少要加 g_i 的油
    - 同时也不会加多于 g_i 的油，因为多出的部分在加油站 u 加更划算
2. 若在 L 公里内，u 是油价最低的一个加油站，但 $p_u > p_i$
    - 加油量 $g_i = L - r_i$，即将油箱加满：假定某一最优方案在 i 加油量少于 g_i，则在下 L 公里中必然要在其他加油站加油，而这些油如果在 i 加显然更划算
    - 下一个要加油的加油站为 u：若在 u 之前或之后停车加油，显然所加的一部分油在 u 加更划算

算法描述

Algorithm `MinTrip`$(p[], d[], n)$

```
1  for k ← 0 to n do g[k] ← 0;
2  r[0] ← 0; cost ← 0; i ← 0;
3  while i < n do
4      D ← 0; v ← u ← i + 1;
5      while D + d[v] ≤ L and v ≤ n do
6          D ← D + d[v];
7          if p[v] ≤ p[u] then u ← v;
8          v ← v + 1;
9          if p[u] ≤ p[i] then break;
10     end
11     if p[u] ≤ p[i] or v > n then  g[i] ← D − r[i]; r[u] ← 0 ;
12     else  g[i] ← L − r[i]; r[u] ← L − Σ_{j=i+1}^{u} d[j] ;
13     i ← u; cost ← cost + p[i] ∗ g[i];
14 end
15 return g and cost;
```

复杂度分析

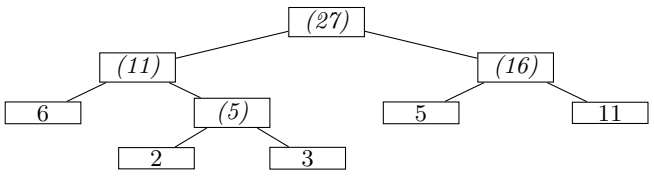
- 在每一个加油站最多需要向后搜索 k 个加油站
- 时间复杂度 $W(n) \in O(kn)$
- 如果 k 为常数，时间复杂度为 $O(n)$
- 如果 k 不是常数，是否有复杂度为 $O(n)$ 的算法？(课后练习)

2.4 钢管焊接问题

问题描述

- 现需要将 n 段钢管 $a_1, a_2, \cdots, a_n$ 焊接成一整条钢管，每次可任选两根钢管焊接在一起，假设钢管 a_i 的重量为 $W[i]$ $(1 \le i \le n)$，将每次焊接代价计为所焊的两根钢管的重量之和。例如，焊接 5 段钢管，其重量分别为 $5, 2, 6, 11, 3$，若按下图所示二叉树的顺序来焊接这些钢管，则焊接总代价为：

$$(2+3)+(5+6)+(5+11)+(11+16)=59$$



- 显然，每一种焊接流程都可以用如上图所示的一棵满二叉树来表示
- 试设计一个贪心算法找出一种最优焊接流程并分析算法的时间复杂度

贪心算法设计

- 每次选取剩余钢管中最轻的两段焊接在一起：

Algorithm `LeastCostWeld`$(W[], n)$

```
1  用 W[1..n] 构造小根堆 H;
2  for i ← 1 to n − 1 do
3  |   x ← ExtractMin(H);
4  |   y ← ExtractMin(H);
5  |   构造结点 z;
6  |   z.left ← x; z.right ← y;
7  |   z.weight ← x.weight + y.weight;
8  |   Insert(H, z);
9  end
10 return Root(H);
```

- 显然，该算法时间复杂度为 $O(n \log n)$

算法正确性分析

Lemma 2.2. $\mathrm{Cost}(T) = \sum_{k=1}^{n} W[k] \cdot \mathrm{depth}(k)$

Proof. 用数学归纳法证明，对二叉树高度 h 归纳：

奠基：$h = 0$ 和 $h = 1$ 时结论显然成立；

归纳：假设对于任意高度小于或等于 h 的二叉树结论成立，则对于高度为 $h+1$ 的二叉树 T，假设其左子树和右子树为 L 和 R，分别包含叶结点 $W[1] \sim W[j]$ 和 $W[j+1] \sim W[n]$，而 T 的根结点代表了最后一步焊接。则根据假设，左子树和右子树的焊接总代价分别为：

$$\mathrm{Cost}(L) = \sum_{k=1}^{j} W[k] \cdot \mathrm{depth}_L(k),\ \mathrm{Cost}(R) = \sum_{k=j+1}^{n} W[k] \cdot \mathrm{depth}_R(k)$$

而其完成的钢管总重量分别为 $\sum_{k=1}^{j} W[k]$ 和 $\sum_{k=j+1}^{n} W[k]$，因此最后一步焊接代价为 $\sum_{k=1}^{j} W[k] + \sum_{k=j+1}^{n} W[k]$，因此，最终的焊接总代价为：

$$\begin{aligned}
\mathrm{Cost}(T) &= \mathrm{Cost}(L) + \mathrm{Cost}(R) + \sum_{k=1}^{j} W[k] + \sum_{k=j+1}^{n} W[k] \\
&= \sum_{k=1}^{j} W[k] \cdot (\mathrm{depth}_L(k) + 1) + \sum_{k=j+1}^{n} W[k] \cdot (\mathrm{depth}_R(k) + 1) \\
&= \sum_{k=1}^{j} W[k] \cdot \mathrm{depth}_T(k) + \sum_{k=j+1}^{n} W[k] \cdot \mathrm{depth}_T(k) \\
&= \sum_{k=1}^{n} W[k] \cdot \mathrm{depth}_T(k)
\end{aligned}$$

因此对于高度为 $h+1$ 的二叉树，结论也成立。 □

Lemma 2.3. 若称焊接总代价最小的焊接流程为最优焊接流程，则在最优焊接流程所对应的满二叉树中，最底层的叶结点中必有一结点代表重量最轻的那段钢管。

Proof. 用反证法证明。

不失一般性，设 $W[1]$ 为最小重量，假设 $W[1]$ 不在二叉树最底层，$W[k]$ 为最底层的一个叶结点，则有 $W[k] > W[1]$，现在交换 $W[1]$ 和 $W[k]$ 的位置，得到新的焊接总代价：

$$\begin{aligned}\mathrm{Cost}(new) &= \mathrm{Cost}(T) - W[1]\,\mathrm{depth}(1) - W[k]\,\mathrm{depth}(k)\\ &+ W[1]\,\mathrm{depth}(k) + W[k]\,\mathrm{depth}(1)\\ &= \mathrm{Cost}(T) - (W[1] - W[k])\,\mathrm{depth}(1) + (W[1] - W[k])\,\mathrm{depth}(k)\\ &= \mathrm{Cost}(T) - (W[1] - W[k])(\mathrm{depth}(1) - \mathrm{depth}(k))\end{aligned}$$

因为 $W[1] < W[k]$ 且 $\mathrm{depth}(1) < \mathrm{depth}(k)$，所以 $(W[1] - W[k])(\mathrm{depth}(1) - \mathrm{depth}(k)) > 0$，所以 $\mathrm{Cost}(new) < \mathrm{Cost}(T)$，这与 T 是最优焊接流程矛盾，故假设不成立，$W[1]$ 在最底层。 □

Lemma 2.4. 在最优焊接流程所对应的满二叉树中，重量第二轻的钢管所对应的叶结点也在二叉树最底层的叶结点中。

Proof. 由于最优焊接流程所对应的二叉树为满二叉树，故其最底层至少有两个叶结点，如果第二轻的钢管不在最底层，则最底层除最轻叶结点外，必有一个结点重量大于第二轻的结点，交换这两个结点的位置，参照前面的推导可得一棵总代价更小的树，因此导致矛盾，原结论成立。 □

Theorem 2.5. 必有一种最优焊接流程在第一步焊接时将最轻的两段钢管焊接在一起。

Proof. 由前面的结论可知最优焊接二叉树最底层的叶结点必包含重量最轻的两段钢管，假设为 $W[1]$ 和 $W[2]$。若二者有共同父结点，则结论成立；若二者父结点不同，则必存在一个叶结点 $W[x]$ 与最轻的结点 $W[1]$ 共有父结点，现交换 $W[x]$ 和 $W[2]$ 的位置，由于都在最底层，结点深度未变，根据第一条引理的结论，焊接总代价也未变，这时 $W[1]$ 和 $W[2]$ 共有父结点，且焊接流程是一种最优焊接流程，因此结论成立。 □

2.5 课后习题

邮局位置问题

Exercise (7). 在一条街上有 n 所房子，$H[i]$ $(1 \le i \le n)$ 是第 i 所房子离街道起点处的距离（以米为单位），假定 $H[1] < H[2] < \cdots < H[n]$。目前该街道上还没有一所邮局，现计划新建若干所邮局，使得每所房子到最近的邮局距离在 100 米以内。试设计一个时间复杂度为 $O(n)$ 的算法，计算出新建邮局的位置，即每所新建邮局离街道起点处的距离 $P[j]$ $(1 \le j \le m)$，同时确保新建邮局个数 m 最小。 *deadline: 2019.12.29*

3 字符串匹配算法

3.1 基本问题

问题描述

- **字符串匹配问题**：在一个字符串（通常称作文本或 text）中搜索给定子串（通常称为模式串或 pattern）
- 字符串匹配问题最早来自文本处理领域，如果将匹配的基本元素从字符扩展到其他元素，则字符串匹配算法可以应用在很多领域的数据检索中，在这一部分只讨论字符串匹配算法
- **记号约定**：

P	待搜索的模式串
T	待搜索文本
m	模式串长度
n	文本长度
p_i, t_i	模式串和文本中的第 i 个字符（下标从 1 开始）
j	文本的当前匹配位置
k	模式串的当前匹配位置

一个简单算法

Algorithm `SimpleScan`$(P[], T[], m)$

```
1  match ← −1; i ← j ← k ← 1;
2  while not EndText(T, j) do
3      if k > m then match ← i; break;
4      if t_j = p_k then j ← j + 1; k ← k + 1;
5      else
6          backup ← k − 1;
7          j ← j − backup; k ← k − backup;
8          i ← j ← j + 1;
9      end
10 end
11 return match;
```

- 最坏情况时间复杂度：$\Theta(mn)$
- 一些统计实验表明，文本 T 中的每个字符的平均比较次数约为 1.1
- 文本匹配位置 j 在扫描过程中需要经常回溯，因此该算法在某些环境下不适用

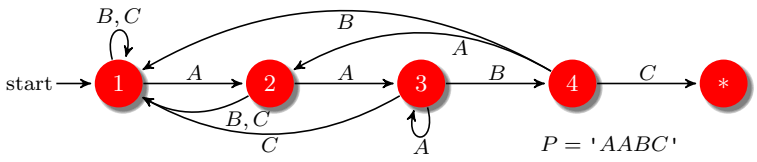
3.2 KMP 算法

模式匹配的有限自动机

- 用于模式匹配的**有限自动机** (finite automaton)：设 Σ 是文本和模式串中可能出现的所有字符的字母表，$\alpha = |\Sigma|$，则有限自动机可用包含下面两种结点的流程图来表示

  1. “读 (read)”结点：表示“读下一个文本字符，如果到达文本末尾，则停机；无匹配结果”
  2. “停 (stop)”结点：表示“停机；有匹配结果”，通常用 $*$ 标记

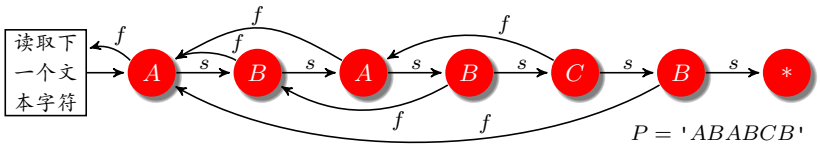
- 该流程图从每个“读”结点发出 α 个箭头，每个箭头标以字母表 Σ 的一个字母，表示匹配该字符所到达的下一个状态



- 一旦构造出与模式串 P 对应的有限自动机，按该自动机构造的算法，文本中的每个字符只需比较 1 次，匹配算法的复杂度可以降到 $O(n)$，最大的困难在于有限自动机的构造

KMP 流程图

- Knuth-Morris-Pratt (KMP) 流程图简化了有限自动机的构造，它与通常的有限自动机有一些区别：
  - 待匹配字符直接标记在结点上而不是箭头上
  - 每个结点只发出两个箭头，分别代表匹配成功和失败
  - 读取下一个字符发生在成功箭头之后
  - 当发生一个失败匹配之后，同一个文本字符将按流程图再次匹配
  - 有一个额外结点表示读取文本中下一个字符
- 例如 $P = 'ABABCB'$，其 KMP 流程图为：



KMP 流程图的构造

- KMP 流程图可用两个数组存储：一个存储模式串（匹配结点），一个存储失败匹配链接（箭头）
- 假设失败匹配链接存储在数组 $fail$ 中，$fail[k]$ 为第 k 个结点匹配失败所指向的下一个结点下标；$fail[1] = 0$

- 设置失败链接 ($fail[7] = 5$)：

$$\begin{array}{|cccccc|cc} A & B & A & B & A & B & C & B \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & & \\ A & B & A & B & A & B & x & \cdots \end{array} \Rightarrow \begin{array}{|cccc|cccc} & & & & A & B & & \\ & & & & & & & \\ A & B & A & B & A & B & x & \end{array} \begin{array}{|cccc} A & B & C & B \\ & & & \\ & & & \cdots \end{array}$$

Definition 3.1 (失败链接). 失败链接 $fail[k]$ 是 $p_1 \cdots p_{r-1}$ 与 $p_{k-r+1} \cdots p_{k-1}$ 匹配的 r ($r < k$) 中的最大值，即：模式串 P 的 $r-1$ 个字符前缀与结束于 p_{k-1} 的 $r-1$ 个字符构成的子串相等

KMP 流程图的构造 (cont.)

- 设置失败链接的递归过程 ($s = fail[k-1]$)：

$$\begin{array}{|cccc|cccc} p_1 & \cdots & p_{k-r+1} & \cdots & p_{k-2} & p_{k-1} & p_k & \cdots & p_m \\ & & \uparrow & \cdots & \uparrow & & & & \\ & & p_1 & \cdots & p_{s-1} & p_s & & \cdots & \\ & & & \cdots & \uparrow & & & & \\ & & & p_1 \quad \cdots & p_{fail[s]-1} & p_{fail[s]} & & \cdots & \end{array}$$

Algorithm `KMPSetup`$(P[], m)$

```
1 fail[1] ← 0;
2 for k ← 2 to m do
3 |   s ← fail[k − 1];
4 |   while s ≥ 1 and p_s ≠ p_{k−1} do s ← fail[s] ;
5 |   fail[k] ← s + 1;
6 end
7 return fail;
```

`KMPSetup` 复杂度分析

- 粗看 `KMPSetup` 最坏情况时间复杂度为 $O(m^2)$，实际上没有这么差
- p_s 和 p_{k-1} 的成功比较最多为 $m-1$ 次
- p_s 和 p_{k-1} 的每一次失败比较导致 s 递减 ($fail[s] < s$)，因此我们可以用 s 值减小的次数来计算失败比较的次数：
  - 当 $k = 2$ 时 s 初始值为 0
  - s 的值当在本次循环执行 $fail[k] \leftarrow s+1$ 及下次循环执行 $s \leftarrow fail[k-1]$ 后增 1，随 for 循环总共 $m-2$ 次
  - s 不可能为负值
- 根据以上分析，s 的值减少次数不可能多于 $m-2$
- 因此 `KMPSetup` 字符比较次数最多为 $2m-3$
- `KMPSetup` 最坏情况时间复杂度为 $O(m)$!

KMP 算法

Algorithm KMPScan($P[], T[], m, fail[]$)

```
1 match ← −1; j ← k ← 1;
2 while not EndText(T, j) do
3 |   if k > m then match ← j − m; break;
4 |   if k = 0 then j ← j + 1; k ← 1;
5 |   else if t_j = p_k then j ← j + 1; k ← k + 1;
6 |   else k ← fail[k] ;
7 end
8 return match;
```

- 时间复杂度：
  - KMPScan 字符比较次数至多为 $2n$ ($Why?$)
  - KMPSetup 和 KMPScan 最坏情况时间复杂度为 $\Theta(n+m)$,比 SimpleScan 的 $\Theta(mn)$ 要好得多
  - 对于自然语言文本，有实验表明 KMP 算法和 SimpleScan 的平均字符比较次数是一样的，但 KMP 算法中不存在文本回溯问题

3.3 BM 算法

基本思路

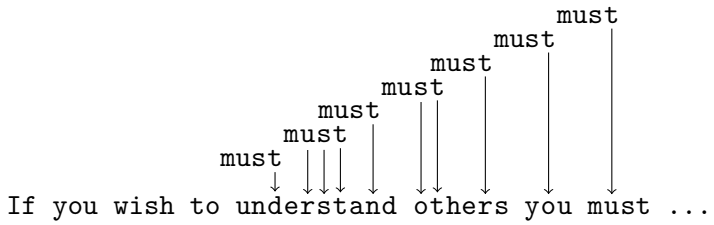
- 无论是 SimpleScan 还是 KMPScan 都需要逐一比较文本串中的字符，但不是所有的比较都是必须的，存在比 KMPScan 更快的算法
- 例如：

```
must must must must
  ↓    ↓    ↓    ↓
If you wish to understand others you must
```

- 模式串越长，所能提供的匹配信息越多
- 一个好的模式匹配算法应该尽可能快地在文本串中跳过不可能出现模式串的部分，避免不必要的比较
- ⇒Boyer-Moore 算法（BM 算法）

改进 1

- 从右向左扫描模式串，直接跳过文本串中不可能产生匹配的部分
- 例如：



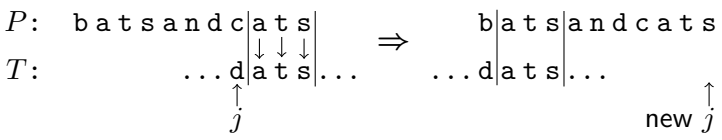
- 计算字符匹配失败情况下，j 在文本串中向后跳跃的字符数：

Algorithm ComputeJumps($P[], m, \Sigma[], \alpha$)

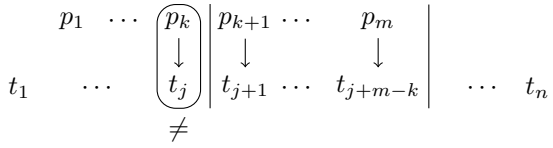
1 for $ch \leftarrow 0$ to $\alpha - 1$ do $charJump[\Sigma[ch]] \leftarrow m$;
2 for $k \leftarrow 1$ to m do $charJump[p_k] \leftarrow m - k$;
3 return $charJump$;

改进 2

- 参考 KMP 算法的失败链接概念，每次失败匹配后可以跳过更多的不可能匹配字符
- 例如：

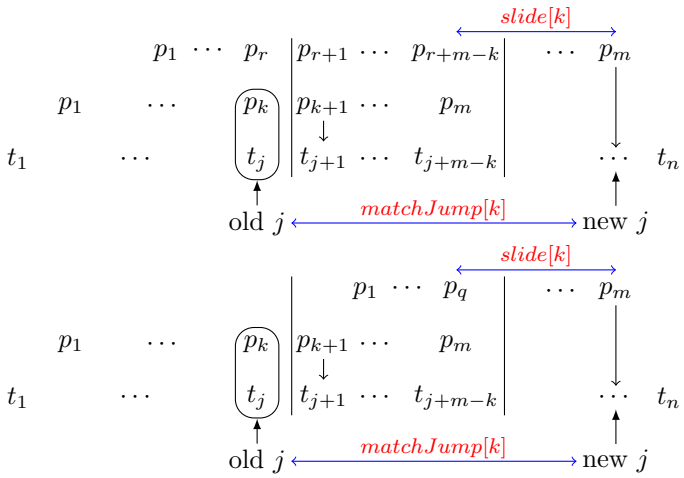


- 一般情况：



改进 2 (cont.)

- 一般情况（续）：



matchJump 和 *slide* 的定义

Definition 3.2 ($matchJump$ 和 $slide$)**.** 首先，$matchJump[k]$ 可根据 $slide[k]$ 计算如下：

$$matchJump[k] = slide[k] + m - k, \quad 1 \le k \le m$$

令 r 为使 $p_{k+1} \cdots p_m$ 和 $p_{r+1} \cdots p_{r+m-k}$ 匹配且 $p_k \neq p_r$ 的最大下标，即：模式串 P 的 $m-k$ 个字符后缀与从 p_{r+1} 开始的 $m-k$ 个字符构成的子串相等。则：

$$slide[k] = k - r, \quad r < k$$

若 $k = m$，则 $p_{k+1} \cdots p_m$ 为空，则 r 为使得 $p_r \neq p_k$ 的最大下标；若找不到子串匹配整个 $p_{k+1} \cdots p_m$，则找模式串 P 前缀和后缀的最长匹配，若该匹配包含 q 个字符，则：

$$slide[k] = m - q$$

计算 *matchJump*

Algorithm ComputeMatchJumps($P[], m$)

```
1  for k ← 1 to m do matchJump[k] ← m + 1 ;
2  sufx[m] ← m + 1;
3  for k ← m − 1 down to 0 do
4  |  s ← sufx[k + 1];
5  |  while s ≤ m and p_{k+1} ≠ p_s do
6  |  |  matchJump[s] ← min(matchJump[s], s − (k + 1));
7  |  |  s ← sufx[s];
8  |  end
9  |  sufx[k] ← s − 1;
10 end
11 low ← 1; shift ← sufx[0];
12 while shift ≤ m do
13 |  for k ← low to shift do
   |  |  matchJump[k] ← min(matchJump[k], shift) ;
14 |  low ← shift + 1; shift ← sufx[shift];
15 end
16 for k ← 1 to m do matchJump[k] ← matchJump[k] + m − k ;
17 return matchJump;
```

ComputeMatchJumps 的正确性？

证明要点：

1. $sufx$ 和 KMP 算法中的 $fail$ 是对称的
2. $sufx[k] = x$ 当且仅当子串 $p_{k+1} \cdots p_{k+m-x}$ 和后缀 $p_{x+1} \cdots p_m$ 匹配
3. 对于 $r_0 = sufx[0]$, $r_{i+1} = sufx[r_i]$，子串 $p_{r_i+1} \cdots p_m$ 是 P 的一个前缀

BM 算法

Algorithm `BMScan`$(P[], T[], m, charJump[], matchJump[])$

```
1  match ← −1; j ← k ← m;
2  while not EndText(T, j) do
3      if k < 1 then match ← j + 1; break;
4      if t_j = p_k then j ← j − 1; k ← k − 1;
5      else
6          j ← j + max(charJump[t_j], matchJump[k]);
7          k ← m;
8      end
9  end
10 return match;
```

一点讨论

- 当 $m \geq 5$ 时，`BMScan` 平均字符比较次数为 cn，其中 $c < 1$；当 $m \leq 3$ 时性能的微弱提升则会被预处理步骤所抵消（主要跟字母表的大小有关）
- 对于自然语言文本，$m \geq 5$ 时有实验表明 BM 算法的平均字符比较次数在 0.24 到 0.3 之间，即文本串中只有 1/4 到 1/3 的字符参与了比较
- 对于二进制流，$charJump$ 的信息已经没有太大意义，此时字符平均比较次数约为 0.7
- 由于 BM 算法采用自右向左的比较方式，因此需要一个字符缓存保存一批待比较的字符
- 参考文献：
  - G. de V. Smit. A Comparison of Three String Matching Algorithms. *Software: Practice and Experience*, Vol.12, No.1, pages 57–66, 1982

3.4 近似匹配算法

基本问题

- 在某些应用环境下不需要或者无法得到精确匹配结果，如单词拼写检查、语音识别等等
- 寻找文本串中模式串的一个近似匹配
- 问题描述：

  Definition 3.3 (k-近似匹配). 给定模式串 $P = p_1 \cdots p_m$ 和文本串 $T = t_1 \cdots t_n$，假定 $n \gg m$，k-近似匹配是指 P 和 T 最多只有 k 个不同之处的匹配。所谓“不同之处”包括以下几种情况（对应于 T 向 P 靠近所需的操作）：

- **修改** (revise) P 和 T 对应字符不同
- **删除** (delete) T 中包含一个 P 中没有的字符
- **插入** (insert) P 中包含一个 T 中没有的字符

需要从 T 中找出给定 P 的 k-近似匹配。

动态规划算法设计

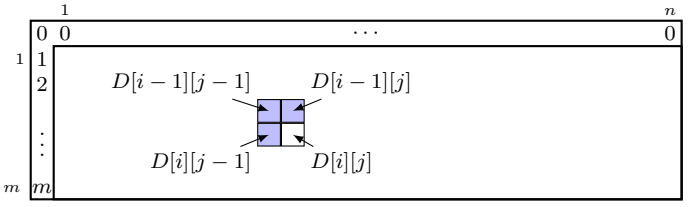
- 在扫描过程中每比较一对字符有四种选择：移动模式串或者执行前面所定义的三种不同情况的对应操作
- 子问题标识：(i, j)
  - 若采用自左向右的扫描模式，i 对应模式串某个后缀的起始位置，j 对应文本串当前匹配的开始位置，子问题可表述为寻找 $p_i \cdots p_m$ 和从 t_j 开始的一段文本的区别最小匹配，则对删除操作来说，子问题 (i, j) 依赖于 $(i, j+1)$，而 $(i, j+1)$ 依赖于 $(i, j+2)$
  - 若采用自右向左的扫描模式，i 对应模式串某个前缀的结束位置，j 对应文本串当前匹配的结束位置，子问题可表述为寻找 $p_1 \cdots p_i$ 和结束于 t_j 的一段文本的区别最小匹配，则对删除操作来说，子问题 (i, j) 依赖于 $(i, j-1)$，而 $(i, j-1)$ 依赖于 $(i, j-2)$
  - 问题的求解顺序一般按 j 的递增顺序，因此自右向左的扫描模式更合适

动态规划算法设计 (cont.)

- 若定义 $D[i][j] = p_1 \cdots p_i$ 和结束于 t_j 的一段文本的最小区别数，则 $D[i][j]$ 是下述四个值中的最小值：

$$
\begin{aligned}
matchCost &= D[i-1][j-1] && \text{若} p_i = t_j \\
reviseCost &= D[i-1][j-1] + 1 && \text{若} p_i \neq t_j \\
insertCost &= D[i-1][j] + 1 && \text{若} p_i \neq t_j \\
deleteCost &= D[i][j-1] + 1 && \text{若} p_i \neq t_j
\end{aligned}
$$

- 任何结束于 t_j 且满足 $D[m][j] \leq k$ 的文本子串都是 k-近似匹配
- 如果只需要找到文本串中出现的第一个 k-近似匹配，则算法只需计算到矩阵 D 最后一行第一个满足上述条件的位置即可停止



算法分析

- 显然，矩阵 D 每个元素的计算开销为常数（包括一个字符比较）
- 因此整个算法复杂度应该在 $O(mn)$，和 `SimpleScan` 一样快
- 空间复杂度：
  - 矩阵 D 大小为 $m \times n$，通常 n 非常大
  - 但算法在计算过程中并不需要存储矩阵 D 中的所有值，而只需要存储当前计算的一列和它的前一列数值
  - 因此空间复杂度为 $O(m)$
- 算法描述 $\Rightarrow$ 课后练习
- 更深入的参考：复杂度为 $O(kn)$ 的 k-近似匹配算法：
  - Gad M. Landau and Uzi Vishkin. Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 220–230