算法设计与分析

动态规划 (I)

目录

算法范式

我们已经学习了两种优雅的设计范式。

- 分治法：将问题分解为独立的子问题，分别求解每个子问题，然后合并子问题的解以形成原问题的解。
- 贪心法：逐步构建解决方案，每次选择能带来最明显和即时收益的下一步。
  - 局部最优选择也能导致全局最优解的问题最适合用贪心法。

算法范式

我们已经学习了两种优雅的设计范式。

- 分治法：将问题分解为独立的子问题，分别求解每个子问题，然后合并子问题的解以形成原问题的解。
- 贪心法：逐步构建解决方案，每次选择能带来最明显和即时收益的下一步。
  - 局部最优选择也能导致全局最优解的问题最适合用贪心法。

这两种范式为各种重要任务产生了大量高效算法。

算法范式

我们已经学习了两种优雅的设计范式。

- 分治法：将问题分解为独立的子问题，分别求解每个子问题，然后合并子问题的解以形成原问题的解。
- 贪心法：逐步构建解决方案，每次选择能带来最明显和即时收益的下一步。
  - 局部最优选择也能导致全局最优解的问题最适合用贪心法。

这两种范式为各种重要任务产生了大量高效算法。

现在我们转向算法领域的另一个重要工具：动态规划，一种适用范围非常广泛的技术。

- 可以预见的是，这种通用性往往以效率为代价。

动态规划历史

动态规划：将问题分解为一系列重叠的同类型子问题，并逐步构建越来越大的子问题的解。

- 本质上是将中间结果缓存到表中以供后续重用的技术

动态规划历史

动态规划：将问题分解为一系列重叠的同类型子问题，并逐步构建越来越大的子问题的解。

- 本质上是将中间结果缓存到表中以供后续重用的技术

Richard Bellman：在 1950 年代开创了动态规划的系统研究。

- dynamic programming = planning over time ⇒ 多阶段过程的最优规划
- 当时的国防部长对数学研究持敌意态度。
- Bellman 为避免冲突而寻求一个令人印象深刻的名称。

[见原文第 7 页图片]

动态规划应用

应用领域

- 生物信息学
- 控制理论
- 信息论
- 运筹学
- 计算机科学：理论、图形学、人工智能、编译器、系统等

一些著名的动态规划算法

- Unix `diff` 命令用于比较两个文件
- Viterbi 算法用于隐马尔可夫模型
- De Boor 算法用于计算样条曲线
- Smith-Waterman 算法用于基因序列比对
- Bellman-Ford 算法用于网络中的最短路径路由
- Cocke-Kasami-Younger 算法用于解析上下文无关文法

目录

DAG 中的最短路径

在有向无环图（DAG）中，从单源节点找最短路径是容易的。令人惊讶的是，它位于动态规划的核心。

- DAG 的节点可以被线性化，即排列在一条线上使得所有边都从左指向右
- 展望未来，通过这种方式我们创建了一个顺序

[见原文第 10 页图：DAG 及其线性化（拓扑排序）]

为什么这有助于最短路径

示例： $s \to d$： 到达 d 的唯一方式是通过其前驱 b 或 c，所以我们只需比较这两条路线：

$$\mathsf{dist}(s, d) = \min\{\mathsf{dist}(s, b) + 1, \mathsf{dist}(s, c) + 3\}$$

为什么这有助于最短路径

示例：$s \to d$：到达 d 的唯一方式是通过其前驱 b 或 c，所以我们只需比较这两条路线：

$$\text{dist}(s,d) = \min\{\text{dist}(s,b)+1, \text{dist}(s,c)+3\}$$

对每个节点都可以写出类似的关系式。

- 按从左到右的顺序计算这些 dist 值 $\Rightarrow$ 在到达节点 v 之前，我们已经有了计算 $\text{dist}(s,v)$ 所需的所有信息 $\Rightarrow$ <u>单遍计算所有距离</u>

DAG 最短路径算法

Algorithm 1 ShortestPath(V, E)

1: 初始化 $\text{dist}(s, v) = \infty$ 对于 $s \neq v$，且 $\text{dist}(s, s) = 0$;
2: **for** $v \in V \backslash s$ 按线性化顺序 **do**
3: $\quad \text{dist}(s, v) = \min_{(u,v)\in E}\{\text{dist}(s, u) + e(u, v)\}$
4: **end for**

DAG 最短路径算法

Algorithm 2 ShortestPath(V, E)

1: 初始化 dist$(s, v) = \infty$ 对于 $s \neq v$，且 dist$(s, s) = 0$;
2: **for** $v \in V \backslash s$ 按线性化顺序 **do**
3:     dist$(s, v) = \min_{(u,v)\in E}\{$dist$(s, u) + e(u, v)\}$
4: **end for**

估计计算复杂度的两种方法:

- 分析算法：最多有 $|E|$ 次比较 $\Rightarrow O(|E|)$
- 分析存储：表dist的大小为 $|V|$，计算每个元素最多需要 $|V|$ 次比较 $\Rightarrow O(|V|^2)$
  - 当图是稀疏图时，第二种估计可能过于粗糙，因为此时 $|E| \ll |V|^2$

总结

上述算法求解了一组子问题

$$\{\mathsf{dist}(s, u)\}_{u \in V}$$

- 从最小的问题开始，$\mathsf{dist}(s, s)$
- 然后逐步求解”更大”的子问题：到线性化中更靠后的顶点的距离
- 大的子问题可以通过之前已解决的较小子问题来求解

总结

上述算法求解了一组子问题

$$\{\mathsf{dist}(s, u)\}_{u \in V}$$

- 从最小的问题开始，dist(s, s)
- 然后逐步求解”更大”的子问题：到线性化中更靠后的顶点的距离
- 大的子问题可以通过之前已解决的较小子问题来求解

这是一种非常通用的技术。

- 在我们的特定情况中 dist$(\cdot, \cdot)$ 计算的是和的最小值，我们同样可以让它变成最大值。
- 或者我们可以用乘积代替求和。

动态规划的关键性质

迭代最优子结构

$\exists$ 子问题的一个排序和一个迭代关系：

- 子问题按排序出现
- 迭代关系展示了如何利用"更小"子问题 P' 的答案来求解子问题 P，即 P 的最优解可以从 $P' \subset P$ 的最优解推导出来

$\rightsquigarrow$ 允许单遍迭代

动态规划范式

动态规划是一种非常强大的算法范式：通过识别一组子问题并逐个解决它们来求解问题

- 最小的子问题先解决
- 利用小问题的答案求解大问题
- 直到解决原问题

在动态规划中，DAG 是隐式的，应该始终牢记

- 节点 $\leftrightarrow$ 子问题/状态（与最优函数值关联）
- 边 $a \rightarrow b$ 表示 a 和 b 之间的依赖关系，换句话说，如果要解决子问题 b 我们需要子问题 a 的答案，那么就有一条（概念上的）从 a 到 b 的边 $\Rightarrow$ a 被认为是比 b 更小的子问题

目录

全源最短路径

现实是复杂的。现实世界需要针对一般有向加权图的算法：G 可能有负权边（但没有负权环）。

- Dijkstra 算法无法处理负权边。
- Bellman-Ford 算法能正确处理一般有向图中的单源最短路径（SSSP），但复杂度较高 $O(|V||E|)$。

如果我们想找到的不仅是从单源 s 出发的最短路径，而是从所有源点出发的最短路径呢?
朴素想法：调用 Bellman-Ford 算法 $|V|$ 次，每个起始节点一次 $\rightsquigarrow$ 运行时间 $O(|V|^2|E|)$

- 通常，$|E| > |V|$

有更好的算法吗?

Floyd-Warshall 算法

Floyd-Warshall 算法：一种更好的动态规划算法，复杂度为 $O(|V|^3)$
基本思想：从 (u, v) 之间的最短路径 $u \to w_1 \to \cdots \to w_l \to v$ 使用若干中间节点——可能没有。

- 假设我们完全禁止中间节点 $\rightsquigarrow$ 一次性解决全源最短路径：$\mathsf{dist}(u, v) = e(u, v)$。

如果我们逐步扩展允许的中间节点集合 S 会怎样?
我们可以每次添加一个节点，在每个阶段更新最短路径长度。

- 最终 S 增长到 $V \Rightarrow$ 此时所有顶点都被允许出现在所有路径上 $\rightsquigarrow$ 找到图中顶点之间的真正最短路径。

基于中间节点的动态规划

将 V 中的顶点编号为 $\{1, 2, \ldots, n\}$，令 $\text{dist}(i, j, k)$ 为从 i 到 j 的最短路径长度，其中只有节点 $\{1, 2, \ldots, k\}$ 可以作为中间节点。

- 初始时，$\text{dist}(i, j, 0)$ 是 i 和 j 之间直接边的长度（如果存在），否则为 ∞。

[见原文第 22 页图]

逐步增加允许的中间节点数量。$\text{dist}(i, j, k)$ 的初始值是 $\text{dist}(i, j, k-1)$。
使用 k 能给我们从 i 到 j 更短的路径当且仅当（k 只出现一次因为没有负权环）

$$\text{dist}(i, k, k-1) + \text{dist}(k, j, k-1) < \text{dist}(i, j, k-1)$$

在这种情况下，$\text{dist}(i, j, k)$ 应相应更新。

Floyd-Warshall 算法

Algorithm 3 FloydWarshall($G = (V, E)$)

for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
        $\text{dist}(i, j, 0) = \infty$
    end for
end for
for $(i, j) \in E$ **do**
    $\text{dist}(i, j, 0) = e(i, j)$
end for
for $k = 1$ **to** n **do**
    for $i = 1$ **to** n **do**
        for $j = 1$ **to** n **do**
            $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k-1) + \text{dist}(k, j, k-1), \text{dist}(i, j, k-1)\}$
        end for
    end for
end for

目录

最长递增子序列

输入：一个数字序列 $a_1, \ldots, a_n$。

- 子序列是这些数字中按顺序取出的任意子集，形如 $a_{i_1}, \ldots, a_{i_k}$，其中 $1 \le i_1 \le \cdots \le i_k \le n$。
- 递增子序列是数字严格递增的子序列。

目标：找到长度最大的递增子序列。

示例

[见原文第 25 页图：序列 5, 2, 8, 3, 6, 9, 7 及箭头]

箭头表示原序列中最优解的连续元素之间的转换。

递增子序列的 DAG

目标：从解空间（所有递增子序列）中找到最优解 $\Rightarrow$ 创建一个包含递增子序列所有允许转换的图

- 为每个元素 a_i 建立一个节点 i，当 a_i 和 a_j 可能成为递增子序列中的连续元素时，添加有向边 (i, j)，即 $i < j \wedge a_i < a_j$

$G = (V, E)$ 是一个 DAG，因为 $(i, j) \in E$ 只在 $i < j$ 时才可能

- 递增子序列与 DAG 中的路径之间存在一一对应

[见原文第 26 页图：序列 5, 2, 8, 3, 6, 9, 7 的 DAG]

动态规划

我们的目标是将 LIS 转化为在 DAG 中找最长路径（每条边权重为 1）。

动态规划

我们的目标是将 LIS 转化为在 DAG 中找最长路径（每条边权重为 1）。定义 $L(j)$：以 j 结尾的最长路径（最长递增子序列）上的节点数

- 将 $L(j)$ 理解为<u>最长路径长度 $+1$</u>，其中 j 为终点，从所有可能的源节点出发

$$\ell = \max_{j \in [n]} L(j)$$

动态规划

我们的目标是将 LIS 转化为在 DAG 中找最长路径（每条边权重为 1）。定义 $L(j)$：以 j 结尾的最长路径（最长递增子序列）上的节点数

- 将 $L(j)$ 理解为<u>最长路径长度 $+1$</u>，其中 j 为终点，从所有可能的源节点出发

$$\ell = \max_{j \in [n]} L(j)$$

为了求解 LIS，我们定义了一组子问题 $\{L(j)\}_{j \in [n]}$，它们具有最优子结构性质，允许单遍求解。

算法与复杂度分析

Algorithm 4 LIS(A)

```
1: for j = 1 to n do
2:     L(j) = 1 + max{L(i) : (i, j) ∈ E}
3: end for
4: return max_j{L(j)}
```

- 注意 $(i, j) \in E$ 只在 $i < j$ 时才可能。

算法与复杂度分析

Algorithm 5 LIS(A)

```
for j = 1 to n do
    L(j) = 1 + max{L(i) : (i, j) ∈ E}
end for
return max_j{L(j)}
```

- 注意 $(i,j) \in E$ 只在 $i < j$ 时才可能。

算法需要知道 j 的前驱

- 构造反向图 G^R 的邻接表（通常在线性时间内）

算法与复杂度分析

Algorithm 6 LIS(A)

```
for j = 1 to n do
    L(j) = 1 + max{L(i) : (i, j) ∈ E}
end for
return max_j{L(j)}
```

- 注意 $(i,j) \in E$ 只在 $i < j$ 时才可能。

算法需要知道 j 的前驱

- 构造反向图 G^R 的邻接表（通常在线性时间内）

$L(j)$ 的计算时间与 j 的入度成正比 $\rightsquigarrow$ 总运行时间与 $|E|$ 成线性关系

- 当输入数组已按递增顺序排列时取得最大值 $\rightsquigarrow$ $W(n) = |E| = O(n^2)$

算法与复杂度分析

Algorithm 7 LIS(A)

```
for j = 1 to n do
    L(j) = 1 + max{L(i) : (i, j) ∈ E}
end for
return max_j{L(j)}
```

- 注意 $(i,j) \in E$ 只在 $i < j$ 时才可能。

算法需要知道 j 的前驱

- 构造反向图 G^R 的邻接表（通常在线性时间内）

$L(j)$ 的计算时间与 j 的入度成正比 $\rightsquigarrow$ 总运行时间与 $|E|$ 成线性关系

- 当输入数组已按递增顺序排列时取得最大值 $\rightsquigarrow$ $W(n) = |E| = O(n^2)$

LIS 对每个" 源" 节点在反向 DAG 中计算最长路径然后选择最大值，而ShortestPath在 DAG 中从一个给定源节点计算到所有其他节点的最短路径。

回溯解

还有最后一个问题需要澄清。

L 值只告诉我们最优子序列的长度，如何恢复子序列本身?

- 这可以通过记录设备轻松管理
  - 在计算 $L(j)$ 时，记录 prev(j)，即到 j 的最长路径上的倒数第二个节点（想想怎么做?）
- 最优子序列可以通过跟踪这些回溯指针来重建。

递归？不，谢谢。

回到我们关于最长递增子序列的讨论

- $L(j)$ 的公式也暗示了一种替代的递归算法。这不是更简单吗?

递归？不，谢谢。

回到我们关于最长递增子序列的讨论

- $L(j)$ 的公式也暗示了一种替代的递归算法。这不是更简单吗?

实际上，递归是一个非常糟糕的想法：所得到的过程将需要指数时间。

- 假设给定的数字是有序的。显然，这是最坏情况。子问题 $L(j)$ 的公式变为：

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$$

递归？不，谢谢。

回到我们关于最长递增子序列的讨论

- $L(j)$ 的公式也暗示了一种替代的递归算法。这不是更简单吗?

实际上，递归是一个非常糟糕的想法：所得到的过程将需要指数时间。

- 假设给定的数字是有序的。显然，这是最坏情况。子问题 $L(j)$ 的公式变为：

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$$

下图展开了 $L(5)$ 的递归。注意同样的子问题被反复求解。

为什么递归不好?

[见原文第 40 页图：$L(5)$ 的递归树]

为什么递归不好?

[见原文第 40 页图：$L(5)$ 的递归树] 节点对应于计算成本。设 $C(n)$ 为 $L(n)$ 的树上的节点数。我们有 $T(n) = C(n)$。

为什么递归不好?

[见原文第 40 页图：$L(5)$ 的递归树] 节点对应于计算成本。设 $C(n)$ 为 $L(n)$ 的树上的节点数。我们有 $T(n) = C(n)$。显然，我们有如下迭代关系：

$$C(n) = C(n-1) + \cdots + C(2) + C(1)$$

- $C(n)$ 关于 n 是指数级的 $\rightsquigarrow$ 递归解是灾难性的

Fibonacci 数的类似情况

[见原文第 44 页图：$F(5)$ 的递归树]

Fibonacci 数的类似情况

[见原文第 44 页图：$F(5)$ 的递归树] 递归方法：复杂度为 $F(n)$。

- 设 $C(n)$ 为 $F(n)$ 的树上的节点数，我们有：

$$C(n) = C(n-1) + C(n-2) = F(n)$$

Fibonacci 数的类似情况

[见原文第 44 页图：$F(5)$ 的递归树] 递归方法：复杂度为 $F(n)$。

- 设 $C(n)$ 为 $F(n)$ 的树上的节点数，我们有：

$$C(n) = C(n-1) + C(n-2) = F(n)$$

迭代方法：复杂度为 $O(n)$。

Fibonacci 数的类似情况

[见原文第 44 页图：$F(5)$ 的递归树] 递归方法：复杂度为 $F(n)$。

- 设 $C(n)$ 为 $F(n)$ 的树上的节点数，我们有：

$$C(n) = C(n-1) + C(n-2) = F(n)$$

迭代方法：复杂度为 $O(n)$。分治方法：复杂度为 $O(\log n)$。

动态规划 vs. 分治法

在分治法领域，问题被表示为显著更小的子问题，比如一半大小。

- 例如，MergeSort通过递归地对两个大小为 $n/2$ 的子数组排序来对大小为 n 的数组排序。
- 由于问题规模急剧下降，完整的递归树只有对数深度和多项式数量的节点。

动态规划 vs. 分治法

在分治法领域，问题被表示为显著更小的子问题，比如一半大小。

- 例如，MergeSort通过递归地对两个大小为 $n/2$ 的子数组排序来对大小为 n 的数组排序。
- 由于问题规模急剧下降，完整的递归树只有对数深度和多项式数量的节点。

在动态规划中，问题被归约为只是稍微小一点的子问题。因此完整的递归树通常具有多项式深度和指数数量的节点。

- 然而，大多数这些节点是重复的 $\rightsquigarrow$ 其中没有太多不同的子问题。
- 因此，效率是通过显式枚举不同的子问题并按正确顺序求解它们来获得的。

目录

最大子段和

问题：给定一个整数数组（可能为负）$A[n]$

$$(a_1, a_2, \ldots, a_n)$$

目标：找到最大子段和：

$$\mathsf{MIS} = \max\left\{0, \max_{1\le i\le j\le n} \sum_{k=i}^{j} a_k\right\}$$

示例：$(-2, 11, -4, 13, -5, -2)$
解：$\mathsf{MIS} = a_2 + a_3 + a_4 = 20$

可能的算法

暴力法：枚举所有可能的 (i,j) 对（$i \le j$），计算和 $a_i + \cdots + a_j$ 并找到最大值。
分治法：将数组分成左半部分和右半部分，分别计算左半部分、右半部分和跨越中间的最大区间，然后找到最大值。
动态规划

暴力算法

分治法

将 $A[n]$ 分成左半部分 $A[1,k]$ 和右半部分 $A[k+1,n]$，其中 k 为中点

- 递归计算 A_L 的 S_L
- 递归计算 A_R 的 S_R

计算以 k 为右边界的最大和 S_1，计算以 $k+1$ 为左边界的最大和 S_2，
输出 $\max\{S_L, S_R, S_1 + S_2\}$

[见原文第 52 页图]

分治算法伪代码

Algorithm 8 MaxIntervalSum($A[i,j]$)

输出：最大区间 MIS 及左/右边界

```
if i = j then
    return max{A[i], 0} 及边界;                // |A| = 1
end if
k ← ⌊(i + j)/2⌋;
S_L ← MaxIntervalSum(A, i, k);
S_R ← MaxIntervalSum(A, k + 1, j);
S_1 ← MaxOneside(A, i, k, ←);
S_2 ← MaxOneside(A, k + 1, j, →);
return max{S_L, S_R, S_1 + S_2} 及边界;
```

- 如果 $A[i] \leq 0$，将左右边界都设为 0
- MaxOneside 的复杂度为 $O(n)$。

$$T(n) = 2\,T(n/2) + O(n) \quad \Big\} \Rightarrow T(n) = O(n \log n)$$

动态规划

子问题：左边界 ≥ 1，右边界为 i
优化函数：$\mathrm{OPT}(i)$ —$A[1,\ldots,i]$ 中必须包含 $A[i]$ 的最大子段和，以 i 为右边界

$$\mathrm{OPT}(i) = \max_{1 \leq k \leq i} \left\{ \sum_{j=k}^{i} A[j] \right\}$$

[见原文第 54 页图]

直接根据定义计算 $\mathrm{OPT}(i)$ 是相当低效的。

优化函数的迭代关系

OPT(i) 的迭代关系：取决于 OPT$(i-1)$ 的贡献

- OPT$(i-1) < 0$：区间仅由 $A[i]$ 组成
- OPT$(i-1) \geq 0$：区间连接到前一个区间

优化函数的迭代关系

$\mathsf{OPT}(i)$ 的迭代关系：取决于 $\mathsf{OPT}(i-1)$ 的贡献

- $\mathsf{OPT}(i-1) < 0$：区间仅由 $A[i]$ 组成
- $\mathsf{OPT}(i-1) \geq 0$：区间连接到前一个区间

$$\mathsf{OPT}(i) = \max\{\mathsf{OPT}(i-1) + A[i], A[i]\}, \quad i = 2, \ldots, n$$

$$\mathsf{OPT}(1) = A[1]$$

$$\mathsf{MIS} = \max_{1 \leq i \leq n} \{\mathsf{OPT}(i)\}$$

伪代码

目录

压缩灰度图像

灰度图像可以看作像素序列（每个像素范围从 $0 \sim 255$，8 位/1 字节）

$$\{a_1, a_2, \ldots, a_n\}, \quad a_i\text{是第}i\text{个像素的灰度值}$$

[见原文第 59 页 Lena 图]

- 这是一张很好的测试图像，因为它包含细节、平坦区域、阴影和纹理。
- Lena Forsén 还是 IEEE ICIP 2015 宴会的荣誉嘉宾，发表了演讲并主持了最佳论文颁奖典礼。

压缩灰度图像

灰度图像可以看作像素序列（每个像素范围从 $0 \sim 255$，8 位/1 字节）

$$\{a_1, a_2, \ldots, a_n\}, \quad a_i\text{是第}i\text{个像素的灰度值}$$

[见原文第 59 页 Lena 图]

- 这是一张很好的测试图像，因为它包含细节、平坦区域、阴影和纹理。
- Lena Forsén 还是 IEEE ICIP 2015 宴会的荣誉嘉宾，发表了演讲并主持了最佳论文颁奖典礼。

定长图像存储：将像素序列化并存储：每个像素占 8 位，n 个像素的图像占 $8n$ 位/n 字节

压缩灰度图像

灰度图像可以看作像素序列（每个像素范围从 $0 \sim 255$，8 位/1 字节）

$$\{a_1, a_2, \ldots, a_n\}, \quad a_i\text{是第}i\text{个像素的灰度值}$$

[见原文第 59 页 Lena 图]

- 这是一张很好的测试图像，因为它包含细节、平坦区域、阴影和纹理。
- Lena Forsén 还是 IEEE ICIP 2015 宴会的荣誉嘉宾，发表了演讲并主持了最佳论文颁奖典礼。

定长图像存储：将像素序列化并存储：每个像素占 8 位，n 个像素的图像占 $8n$ 位/n 字节

观察到图像通常有一些局部模式。有更好的存储方法吗?

变长压缩

变长压缩格式：用变长编码灰度值以节省存储：将 $\{a_1, a_2, \ldots, a_n\}$ 分成 m 个段：$S_1, S_2, \ldots, S_m$

[见原文第 62 页图：$S_1, S_2, \ldots, S_m$]

S_k 包含 ℓ_k 个像素，S_k 中的像素最多占 b_k 位

$$b_k = \max_{a \in S_k}\{\lceil \log(a+1) \rceil\}$$

- 固定 S_k 的最大长度为 256 $\Rightarrow$ ℓ_k 可以用 8 位表示
- S_k 的 b_k 在 $[1, 8]$ 之间 $\Rightarrow$ b_i 可以用 3 位表示
- S_k 的头部：$\ell_k + b_k = 11$ 位 $\rightsquigarrow$ 解码所需

$$\text{总存储} = \sum_{k=1}^{m}(b_k \cdot \ell_k + 11)$$

压缩灰度图像

约束：

- 第 k 段的长度：$\ell_k \leq 256$
- 第 k 段占用：$b_k \times \ell_k + 11$
- $b_k = \lceil \log(\max_{a \in S_k} + 1) \rceil \leq 8$

目标：给定 $\{a_1, a_2, \ldots, a_n\}$，找到最优划分：

$$\min_{P} \left\{ \sum_{k=1}^{m} (b_k \times \ell_k + 11) \right\}$$

$$P = \{S_1, S_2, \ldots, S_m\} \text{是一个划分}$$

示例

灰度值序列

$$\{10, 12, 15, 255, 1, 2, 1, 1, 2, 2, 1, 1\}$$

① $S_1 = \{10, 12, 15\}$, $S_2 = \{255\}$, $S_3 = \{1, 2, 1, 1, 2, 2, 1, 1\}$

$$11 \times 3 + 4 \times 3 + 8 \times 1 + 2 \times 8 = 69$$

② $S_1 = \{10, 12, 15, 255, 1, 2, 1, 1, 2, 2, 1, 1\}$

$$11 \times 1 + 8 \times 12 = 107$$

③ $S_1 = \{10\}$, $S_2 = \{12\}$, $S_3 = \{15\}$, $S_4 = \{255\}$, $S_5 = \{1\}$, $S_6 = \{2\}$, $S_7 = \{1\}$, $S_8 = \{1\}$, $S_9 = \{2\}$, $S_{10} = \{2\}$, $S_{11} = \{1\}$, $S_{12} = \{1\}$

$$11 \times 12 + 4 \times 3 + 8 \times 1 + 1 \times 5 + 2 \times 3 = 163$$

结论：第一种划分更好

动态规划方法

子问题：左边界始终为 1，右边界为 i

- 像素序列：$\{a_1, a_2, \ldots, a_i\}$
- 优化函数：OPT(i) 是 $\{a_1, \ldots, a_i\}$ 的最小存储位数

计算顺序

[见原文第 65 页图：$i = 1, i = 2, \ldots, i = n$ 的示意]

算法设计

OPT(i)：$\{a_1, a_2, \ldots, a_i\}$ 的最优存储。设 S_m 为最后一段，ℓ_m 为其长度。OPT 的迭代关系为：

$$\text{OPT}(i) = \min_{1\leq\ell_m\leq\min\{i,256\}} \{\text{OPT}(i-\ell_m) + \ell_m \times b_m + 11\}$$

$$b_m = \left\lceil \log\left(\max_{a\in S_m}\{a\}\right)\right\rceil \leq 8$$

$$\text{OPT}(0) = 0$$

[见原文第 66 页图]

算法

演示

输入：$I = \{10, 12, 15, 255, 1, 2\}$。假设我们已经完成了右边界到 $i = 5$ 的子问题计算。

i	1	2	3	4	5	6
$\mathsf{OPT}(i)$	15	19	23	42	50	?
$L(i)$	1	2	3	1	2	?

演示

[见原文第 74 页图]

回溯最优解

Algorithm 9 Traceback($L(n)$)（输入是回溯表）

输出：最优划分 P

```
k ← 1;
while n ≠ 0 do
    P(k) ← L(n);
    n ← n − L(n);
    k ← k + 1;
end while
反转 P;
```

- $P(k)$：第 k 段的长度
- 复杂度：$O(n)$