CS15210 Lab

师宇哲

December 2019

目录

摘要

这是 $CS15210: Parallel\ and\ Sequential\ Algorithms\ and\ Data\ Structures\ Lab$ 实验报告。整个实验课程共有九个实验。对于每个实验，我描述了问题内容，部分进一步分析了问题可以等价为哪类已知问题；我用自然语言描述了算法的思想，对于一些需要重要理论基础的算法，补充说明了相关引理并给出了证明；对于一些复杂算法我给出了收敛性和正确性的证明；给出了 SPARC 风格的伪代码详述算法步骤；大部分实验都思考了多种解法并且在分析算法复杂度的部分给出了对比，绘制了它们的渐进复杂度函数图象对比；每个实验都有一组测试样例来说明算法实际运行的过程；对于一些实现中需要特别注意的算法，我在实现细节中予以特别说明。

第一章　简介

这是 $CS15210: Parallel\ and\ Sequential\ Algorithms\ and\ Data\ Structures\ Lab$ 实验报告。整个实验课程共有九个实验，包括：无重复排序、单源最短路、最大括号距离、天际线问题、括号匹配、高精度运算、割点割边的判定、区域最值查询和素性测试。

对于每个实验，我首先描述了问题内容，对于较抽象的问题，还进一步分析了问题可以等价为哪类已知问题。随后我用自然语言描述了算法的思想，对于一些需要重要理论基础的算法，我补充说明了相关引理并给出了证明；对于一些复杂算法我给出了收敛性和正确性的证明；此外，每个实验的算法我都给出了 SPARC 风格的伪代码详述算法步骤。本着一题多解的原则，我大部分实验都思考了多种解法并且在分析算法复杂度的部分给出了对比。直观起见，我绘制了它们的渐进复杂度函数图象对比。每个实验都有一组测试样例来说明算法实际运行的过程；对于一些实现中需要特别注意的算法，我在实现细节中予以特别说明。

下面简要介绍我在九个实验实验中的工作。实验 1 使用快速排序实现，我对比分析了三种主元选取策略带来的复杂度差异；实验 2 使用 Dijkstra 算法求解，对比了其与 Bellman-Ford 算法的应用范围差异和复杂度差异；实验 3 使用串行栈求解，对比了递归的分治策略；实验 4 使用基于大顶堆的分治算法实现，对比了暴力扫描线方法；实验 5 使用串行栈迭代，对比了可并行化的基于 $Scan$ 的方法；实验 6 使用竖式计算暴力求解，讨论了可以分治策略下并行的大数乘法；实验 7 使用基于 DFS 的 Tarjan 算法实现；实验 8 使用 Sparse Table 方法求解；实验 9 使用 Miller-Rabin 算法实现素性测试，使用线性筛法实现对给定序列的素数筛选，对比了暴力枚举法。

本实验报告后面的内容开展如下：尽管不是必须，但为了保证严谨性，我在第二章中回顾了实验所需的理论基础，包括并行计算的理论基础和分析模型、算法复杂度估计方法、三种 SML 风格的基础数据结构和一个说明算法证明策略的例子；随后，从第三章到第十一章对应九个实验；最后第十二章是实验课总结与心得体会。

第二章　理论基础回顾

本章回顾实验课程中涉及到的理论基础，包括九个实验算法相关正确性证明和复杂度分析的数学前提，以及描述算法的基础抽象数据结构。

2.1　PRAM 模型：并行程序的抽象

PRAM 是 *Parallel Random Access Machine* 的缩写，是我们研究并行程序的抽象模型。它由 $Random\ Access\ Machine, (RAM)$ 扩展而来，一个 PRAM 机器含有 p 个串行的 RAM 机器，它们拥有共享内存。由于 PRAM 模型在描述例如分治法等需要非对称地分配处理器的算法时非常复杂，接下来的报告中我们还是使用一种基于语言的模型，$A\ Strict\ Language\ for\ Parallel\ Computing, (SPARC)$，结合 $Work - Span$（下一小节回顾）模型来分析算法。

2.2　Work *and* Span: 算法复杂度分析方法

2.2.1　渐近分析

在算法复杂度分析中，通常我们不会写出精确的多项式表达，而是用近似的方式使用较简单的表达式表示，即渐近分析。简单来说，它能够定性地说明随着 n 规模的增大，复杂度表达式的增长速度不同。

$$g(n) = O(f(n)) \Rightarrow g(n) \leq c \times f(n)$$

由此可见，大 Oh 标记确定了复杂度的上界，即 g(n) 的增长速度不会大于 f(n) 的增长速度。

$$g(n) = \Omega(f(n)) \Rightarrow g(n) \geq c \times f(n)$$

大 Omega 标记确定了复杂度的下界，即 g(n) 的增长速度不会小于 f(n) 的增长速度。

$$g(n) = \Theta(f(n)) \Rightarrow c_1 \times f(n) \leq g(n) \leq c_2 \times f(n)$$

大 Theta 标记是一个较为严格的标记，它确定了复杂度的上界和下界。

$$g(n) = o(f(n)) \Rightarrow g(n) < c \times f(n)$$

$$g(n) = \omega(f(n)) \Rightarrow g(n) > c \times f(n)$$

而以上这两个小写标记只是其对应的大写标记的严格不等于写法。

实际分析中，最常用的还是大 Oh 标记，因为逻辑上讲，我们对算法的期望是它比最坏情况好一些即可。

2.2.2 Work *and* Span

在算法复杂度分析中，$work$ 指算法运行中基础操作进行的总次数。对于串行算法而言，这就代表了其运行时间。对于我们的并行算法，由于操作可以分配给不同处理器同时进行，实际需要的时间要少一些。假设我们有 P 个处理器执行操作总次数为 W 的程序，$\frac{W}{P}$ 就是理论上的**最佳并行率**。显然地，$\frac{W}{P}$ 越小，越接近理想情况，当处理器趋近于无穷多时，而算法在有限次操作内能完成，则最佳并行率趋近于零。

在并行程序中，我们引入了 $span$ 的概念，它与并行程序中最长的一条只能串行执行的序列长度有关，当理想化地认为我们有无数处理器时，$span$ 可用来表示并行程序运行时间。在我们的实验中，**所有的算法复杂度分析都基于这种假设**。

对串并行程序，我们在通过两个部分的复杂度计算总复杂度时，遵循不同的策略 [表 2.1]。例如，在并行算法中 $MergeSort$ 有 $W = \Theta(nlgn), S = \Theta(lg^2n)$. 它的最佳并行率就是 $\Theta(n/lgn)$. 这个结论可以推广到所有以比较为基本操作的并行排序算法中。

	Work	Span
串行程序	$1 + W_1 + W_2$	$1 + S_1 + S_2$
并行程序	$1 + W_1 + W_2$	$1 + \max\{S_1, S_2\}$

表 2.1: 串并行程序复杂度分析方法对比

2.2.3 $Work - Span$ 模型

在此模型中首要的概念就是**平均并行度**，定义为

$$\overline{P} = \frac{W}{S}$$

这引出了**贪心规划**，即如果同时有一个闲置处理器和待处理任务，则这个闲置处理器立刻开始执行待处理任务。因此，我们得到了贪心规划原则

$$T_P < \frac{W}{P} + S$$

其中 T_P 是程序运行时间。这个不等式给出了贪心规划下程序运行时间的上界。

证明. 首先，最理想情况下我们可以把所有工作平均地分配到所有处理器上，因此，运行时间不可能低于 $\frac{W}{P}$。其次，工作时间不可能低于最长的串行执行序列，于是

$$T_P \geq \max(\frac{W}{P}, S)$$

由此可见，贪心规划已经接近最好情况。实际上 $\frac{W}{P}+S$ 不可能大于 $\max(\frac{W}{P},S)$. 在理想情况之外，我们可以引进平均并行度，将 S 替换为 $\frac{W}{P}$

$$\begin{aligned} T_P &< \frac{W}{P}+S \\ &= \frac{W}{P}+\frac{W}{\overline{P}} \\ &= \frac{W}{P}(1+\frac{P}{\overline{P}}) \end{aligned} \tag{2.1}$$

因此，当并行度远超处理器个数时，T_P 可以逼近理想并行率 $\frac{W}{P}$. □

我对此的理解是**我们可以通过设计良好的处理器规划策略，尽可能地用较少的处理器个数实现处理器个数较多时的使用效果**。

最后，我们定义加速，这是串行序列执行时间 T_s 与总时间的比值。

$$S_P = \frac{T_s}{T_P}$$

2.2.4 递归型算法复杂度分析策略

由于 SPARC 函数不依赖副作用，且程序中不含显式循环体，因此许多函数都以递归定义的方式表示，分析含有这类函数的算法的复杂度时需要求解递归式。递归式复杂度通常结合着分治算法，有以下的形式

$$W(n) = 2W(n/2) + O(n)$$

其中 O(n) 代表了一系列不超过一阶的多项式。我们通过多种策略求解。

第一种是 $Tree\ Method$，首先把原式改写为

$$W(n) \leq 2W(n/2) + c_1 \times n + c_2$$

则随着树结构的展开，设层数为 k，则对于第 k 层的 2^k 个结点，每个结点的多项式改写为

$$c_1 \times (n/2^k) + c_2$$

这一层每个结点多项式和最多（没有空结点）为

$$c_1 \times n + 2^k \times c_2$$

由于树的高度就是 n 的对数 lgn，我们可以得到

$$\begin{aligned} W(n) &\leq \sum_{i=0}^{lgn}(c1 \times n + 2^i \times c_2) \\ &= c_1 n(1+lgn) + c_2(n+\frac{n}{2}+\frac{n}{4}+\cdots+1) \\ &= c_1 n(1+lgn) + c_2(2n-1) \\ &\in O(nlgn) \end{aligned} \tag{2.2}$$

第二种是 $Brick\ Method.$ v 是树结构中一个结点，$N(v), C(v), D(v)$ 分别是它的输入规模，执行开销和子结点集合。$\alpha > 1, a \geq 1$, 下面分三种情况讨论。它们共同的前提条件都是对于结点 $v, N(v) > a$.

1. *Root Dominated*

$$C(v) \geq \alpha \sum_{u \in D(v)} C(u)$$

一言以蔽之，如果父结点的开销至少常数倍大于其所有子结点开销之和，总开销就由根支配，即由递归表达式的“多项式部分（后半部分）”决定。例如：

$$W(n) = 2W(n/2) + n^2$$

由于父结点开销为 n^2，而子结点开销之和为 $n^2/4 + n^2/4 = n^2/2 < n^2$，故总开销由根支配，

$$W(n) \in O(n^2)$$

2. *Leaf Dominated*

$$C(v) \leq \frac{1}{\alpha} \sum_{u \in D(v)} C(u)$$

如果所有子结点开销之和至少常数倍大于其父结点开销，总开销就由叶子支配，即由递归表达式的“递归部分（前半部分）”决定。本质上，递归部分决定了叶子的个数，即总开销由叶子个数决定。例如：

$$W(n) = 2W(n/2) + \sqrt{n}$$

由于父结点开销为 $\sqrt{n}$，而子结点开销之和为 $\sqrt{n/2} + \sqrt{n/2} = \sqrt{2} \times \sqrt{n} > \sqrt{n}$，故总开销由叶子支配，叶子数量为 $2^{lgn} = n$，每个叶子开销为基础情况，即 1. 故

$$W(n) \in O(n)$$

3. *Balanced* 当以上两种情况均不满足时，考虑平衡情况。这时总开销的上界就是递归树的层数与每一层中最大开销之乘积。例如：

$$W(n) = 2W(n/2) + c_1 n + c_2$$

父结点开销为 $c_1 n + c_2$，而子结点开销之和为 $c_1 n/2 + c_2 + c_1 n/2 + c_2 = c_1 n + 2c_2$，父结点和子结点多项式高阶系数相同，因此可以认为是渐进相等的，故属于平衡情况。由于每层中最大的的开销上界为 $(c_1 + c_2)n$, 共有 $(1 + lgn)$ 层，于是有

$$W(n) \in O(nlgn)$$

总之，*Brick Method* 是非常强大的方法，它能够弥补主定理在使用范围中的限制缺陷。

第三种是 *Substitution Method*，适用于带有高阶多项式的根结点开销的递归式，形如

$$W(n) = 2W(n) + kn^{1+\epsilon}$$

令 $\kappa = \frac{k}{1-1/2^\epsilon}$，则

$$\begin{aligned} W(n) &\leq 2W(n) + kn^{1+\epsilon} \\ &= 2\kappa(\frac{n}{2})^{1+\epsilon} + kn^{1+\epsilon} \\ &= \kappa n^{1+\epsilon} + (2\kappa(\frac{n}{2})^{1+\epsilon} + kn^{1+\epsilon} - \kappa n^{1+\epsilon}) \\ &\leq \kappa n^{1+\epsilon} \end{aligned} \tag{2.3}$$

于是，

$$W(n) \in O(n^{1+\epsilon})$$

至此，三种递归式算法复杂度分析策略回顾完毕。*Tree Method* 是 *Brick Method* 的基础，而 *Brick Method* 适用于任何形式的递归式，只需要判断某一个父结点开销与其所有子结点开销之和的大小关系即可求解，*Substitution Method* 适用于带有高阶多项式的根开销的递归式，其实代表了 *Brick Method* 三种情况中根结点支配的情况。

2.3 三种抽象数据类型

2.3.1 Sequence: **顺序结构**

Sequence 是最基础的数据结构，它是一个从自然数集到域 $\{0, 1, \ldots, n-1\}$ 的映射。在这部分中，我将回顾 Sequence 的所有操作并分析它们的开销。由于顺序结构有数组、平衡二叉搜索树、链表三种实现方式，开销的分析将从这三个方面进行。值得注意的是，并行度较高的是平衡二叉搜索树，而链表实现几乎只能串行执行。由以下几种基本函数构成

- *Length*: 输入一个串，返回其长度即元素个数。
- *Nth*: 输入一个索引，返回对应位置元素，若访问越界则抛出错误。
- *Empty*: 返回一个空串。
  IsEmpty: 输入一个串。判断其是否为空。
- *Singleton*: 输入一个元素，构造一个只有该元素的长度为 1 的串。
  IsSingleton: 输入一个串，判断其是否只有一个元素。

以上 6 种操作，除了 *Nth* 之外，三种实现方式中 W,S 均为 1。*Nth* 在 Array 中直接访问索引，W,S 均为 1，在搜索树中二分查找，W,S 均为 $lg|s|$，在链表中顺序查找，W,S 均为 N。

除此之外，串有许多具有 SML 语言特性的操作。

- *Tabulate*: 输入一个函数和自然数 n, 返回一个长度为 n 的序列，其索引为 0 n, 由对自然数 0 n 分别输入函数得到的输出得到，可并行执行。

$$tabulate\ f\ n = \langle f(0), f(1), \ldots, f(n-1) \rangle$$

- *Map*: 输入一个函数和长度为 n 的序列，返回一个对序列每个元素都分别施行函数操作的与原序列长度相等的序列。可并行执行。实际上，*Map* 就是 *Tabulate* 的扩展，后者只能生成对一个从 0 开始单调递增序列的处理后序列，而前者可以对任何序列施行这种操作。

$$map\ f\ \langle a_0, \ldots, a_{n-1}\rangle = \langle f(a_0), \ldots, f(a_{n-1})\rangle$$

- *Filter*: 输入一个布尔型函数和长度为 n 的序列，返回一个只包括输入函数后返回 TRUE 的元素，前后序列长度可能不相等。

$$filter\ f\ S = \{a \in S | f(a) = True\}$$

这三种操作对于 Array 和 BST 都是可以并行执行的，因此 Work 都是和式 $1 + \sum_{x\in s} W(f(x))$(其中 *Tabulate* 应该是从零开始的 n 个自然数相加，方便起见与后两种操作统一，下同)。对于 Array，*Filter* 由于最后要合并结果，Span 的常数项变成对数项，为 $lg|s| + \max_{x\in s} S(f(x))$，另外两种都是最大值式 $1 + \max_{x\in s} S(f(x))$。对于 BST，三种操作 Span 都多了一个对数项，是 $1 + lg|s| + \max_{x\in s} S(f(x))$。而 List 中 W,S 相等都是和式。

对于操作子串，有取出首元素的 *splitHead*，取出尾元素的 *splitTail*，取出以第 k 项为末尾的子串 *take k*，取出以第 k 项尾开头的子串 *drop k*。

接下来的函数以多个串作为输入。

- *Subseq*: 输入一个串和一个数对，返回索引以数对为界的子串。

$$subseq\ s\ (a, b) = s[a \ldots b]$$

在 Array 中 W,S 都是 1，
BST 中 W,S 都是 $1 + lg|s|$，
List 中 W,S 都是 $1 + a$.

- *Append*: 输入两个串，将后一个顺序接在前一个之后。

$$append\ a\ b = a + +b$$

在 Array 中 $W = 1 + |a| + |b|, S = 1$，
BST 中 W,S 都是 $1 + |lg(|a|/|b|)|$。
List 中 W,S 都是 $1 + |a|$.

- *Flatten*: 输入一个以串作为元素的串，类似二维数组，返回一个元素为所有子串中元素按顺序排列的串。

$$flatten\ \langle S_0, \ldots S_{n-1}\rangle = S_1 + +S_2 + + \cdots + +S_{n-1}$$

Array: $W = 1 + lg|s| + \sum_{x\in s} |s|, S = 1 + lg|s|$,
BST: $W = 1 + |s|lg(\sum_{x\in s} |x|), S = 1 + lg(|s| + \sum_{x\in s} |x|)$,
List: $W = S = 1 + |s| + \sum_{x\in s} |x|$.

接下来的函数可以针对特定位置的元素进行操作。

- *Update*: 输入一个串，索引 k，和要更新的值 x，则返回一个将第 k 个元素之替换为 x 的等长度串。

$$update\ s\ k\ x = s' : s[k] \leftarrow x$$

Array: $W = 1 + |s|, S = 1$,
List:$W = S = 1 + |s|$

- *Inject*: 输入一个串和一个索引-更新值的键值对作为元素的串，实际上就是 *update* 的批量操作版本。

$$inject\ s\ \langle (k_0, x_0), \ldots, (k_{m-1}, x_{m-1}) \rangle = s' : (s[k_0] \leftarrow x_0, \ldots, s[k_{m-1}] \leftarrow x_{m-1})$$

Array: $W = 1 + |s| + |m|, S = 1$,
BST: $W = 1 + (|s| + |m|)lg|s|, S = 1 + lg(|s| + |m|)$,
List: $W = S = 1 + |s| + |m|$.

接下来的函数用于合并拥有相同键的键值对。

- *Collect*: 输入一个确定比较规则的比较函数，一个以键值对作为元素的串，函数合并所有拥有相同键的键值对，返回一个以键和它对应的所有元素组成的序列组成的键值对为元素的串。

$$collect\ cmp; \langle (k_0, v_0), (k_0, v_1), \ldots \rangle = \langle (k_0, \langle v_0, v_1, \ldots \rangle), (k_1, \langle v_3, v_4, \ldots \rangle), \ldots \rangle$$

Array, BST: $W = 1 + W(f)|s|lg|s|, S = 1 + S(f)lg^2|s|$,
List: $W = S = 1 + W(f)|s|lg|s|$.

接下来的序列元素会得到扩展，不只是以数为基础的，也可以是一些列操作或步骤。

- *Iterate*: 输入一个对处在某状态的元素进行操作的函数，初始状态和一个串。在每一步中，函数对上一步操作得到的状态和上一步操作得到的值进行操作，得到新的状态和值，将其作为下一步操作的初始状态和初始值，返回最终结果。

$$iterate\ f\ x\ s = \begin{cases} x & |\text{s}|{=}0 \\ f(x, s[0]) & |\text{s}|{=}1 \\ iterate\ f\ (f(x, s[0]))\ (s[1 \ldots |s| - 1]) & \text{otherwise} \end{cases} \tag{2.4}$$

三种实现均为 $W = 1 + \sum W(f(y, z)), S = 1 + \sum S(f(y, z))$.

- *IteratePrefixes*: 操作和上一个函数完全相同，只是返回值除了有结果之外，还有一个长度为 $|s| - 1$ 的串，记录了每一次迭代得到的中间结果，这两个结果组成一个数对返回。

在回顾下一个操作前，我要补充定义一个数学前提。如果函数 f 满足 $f(f(x, y), z) = f(x, f(y, z))$，则这个函数具有结合性 (Associativity)。这种操作只适用于具有结合性的函数。

- *Reduce*: 输入一个串，满足结合律的函数，一个左值 id（与 $Iterate$ 中的初始状态等价），进行类似分治递归的操作，返回最终结果。事实上，当输入参数相同时，它的结果与 $Iterate$ 相同。而 $Reduce$ 的优势在于不依赖于中间步骤的结果，这为并行化提供了可能。

$$reduce\ f\ id\ s = \begin{cases} id & |s|=0 \\ s[0] & |s|=1 \\ f\ (reduce\ f\ id\ (s[0 \ldots \lfloor \frac{|s|}{2} \rfloor - 1]), reduce\ f\ id\ (s[\lfloor \frac{|s|}{2} \ldots |s| - 1])) & \text{otherwise} \end{cases} \tag{2.5}$$

  Array, BST: $W = 1 + \sum W(f(y,z)), S = lg|s| \max S(f(y,z))$,
  List: $W = S = 1 + \sum W(f(y,z))$.

- *Scan*: 输入与 $Reduce$ 一致。不同的是，$Scan$ 兼具了能使用中间结果和并行操作的优势——解决方案就是使用前缀和。函数对于某元素，函数对其所有不包括当前元素的从串首开始的前缀子串分别进行操作，得到中间结果，由此可见，本操作可以并行实现。最后返回结果形式与 $IteratePrefixes$ 相同，一个长度 $|s|-1$ 的中间结果串和最终结果，这个最终结果与相同参数进行 $Reduce$ 操作一致。事实上，$Reduce$ 的并行化是基于分治策略，而 $Scan$ 的并行化是基于可并行迭代的策略。分治和迭代是实验程序设计中都要大量使用的技巧。

$$scan\ f\ id\ s = (\langle reduce\ f\ id\ s[0 \ldots (i-1)] : 0 \le 1 < |s| \rangle, reduce\ f\ id\ s)$$

  Array, BST: $W = |s|, S = lg|s|$,
  List: $W = S = |s|$.

- *ScanI*: 与 $Scan$ 的区别仅在于前缀子串包含当前元素，因此，返回结果只是一个前缀结果串，最后一个元素就是最终结果。

$$scani\ f\ id\ s = \langle reduce\ f\ id\ s[0 \ldots (i)] : 0 \le 1 \le |s| \rangle$$

实际操作中还可以使用单线程序列 ($Single-Threaded\ Sequence$)，在树和图中常用。它的特点是一次操作只能修改一个值以避免冲突，所有每一个历史版本都会保留，并且按照从新到旧的顺序被赋予递增的版本编号并排列在一个链表中，链表头的位置是最新版本。

2.3.2 Binary Search Tree: **平衡搜索树结构**

满二叉树：一棵只含有叶子和内点的树。前者是没有任何孩子结点的结点，后者是拥有左孩子和右孩子的结点。

平衡搜索树：BST 是一个有序集，满足其每个内点到集合中均有一对一映射，且该内点左子树中每个元素的 Key 小于该内点的 Key，右子树每个元素的 Key 大于该内点的 Key。

平衡搜索树的所有操作核心就是把一棵树视为三个参数：$Left(t_1), Root(k), Right(t_2)$。需要强调的是，操作后的左右子树、合并成的树、拆分成的树仍然保持着平衡搜索树的性质，这就意味着不可能简单地对根结点操作了事，而是要重新改变子树的形态。原先一侧的某个子树在操作后转移到另一侧

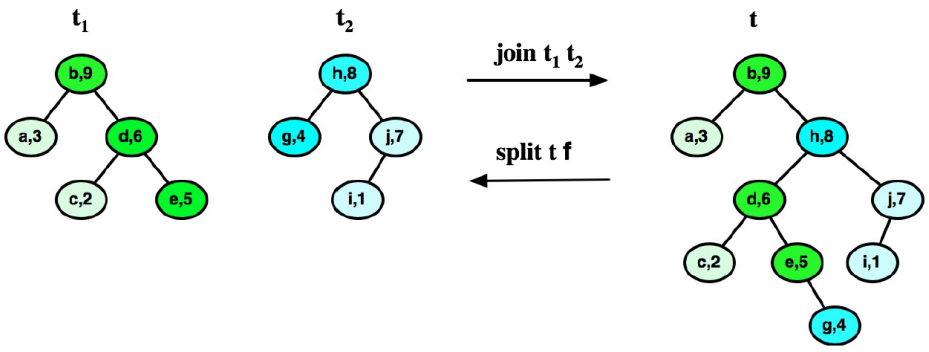

图 2.1: Treap 的对易操作

是常见的。任何操作都是在这三个参数的基础上完成的，它们也是考虑树结构的基本元素。下面有关时间复杂度的分析规定 $n = max(|t_1|, |t_2|), m = min(|t_1|, |t_2|)$。

- $split\ t\ k$：在给定结点将树分成两个子树

$$W = S = O(lg|t|)$$

- $join\ t_1\ t_2$：寻找两个数的共同根，从那里将树合并

$$W = S = O(lg(|t_1| + |t_2|))$$

本操作与上一个互易。

- $find\ t\ k,\ insert\ t\ k,\ delete\ t\ k$：均为针对树元素的操作

$$W = S = O(lg|t|)$$

- $intersect\ t_1\ t_2,\ union\ t_1\ t_2, difference\ t_1\ t_2$：对两棵树取交、并、差集，非常直观，新树的元素必须满足这些集合运算，且满足 BST 性质。

$$W = O(mlg(\frac{n}{m})),\ S = O(lgn)$$

下面介绍 Treap，这是一种带优先级的堆，满足父亲结点的优先级高于其任何一个子结点。因此，Treap 的调用形式是四个参数，l, k, p, r。Treap 有一些与 BST 相近的操作，例如对易的 join 和 split[图 2.1]。前者操作复杂度依然是 $O(lg(|h_1| + |h_2|))$，后者是 $O(lg(h))$。Treap 相当平衡，其高度为 $O(lgn)$。Treap 可以用于带优先级的快速排序，这一点会在实验 1 中涉及。

此外，还有结点数据得到增广的树，它拥有斐波那契性质，每个父亲结点的值是其子树结点值（包括自己）之和。它的操作有 $rank\ t\ k,\ select\ T\ i$，复杂度满足 $W = S \in O(lgn)$。

2.3.3 Sets *and* Table: **集合与表**

Set 集合的数学色彩非常浓厚，基于集合的操作非常容易理解，它有 ArraySequence 和 Tree 两种实现方式。

- $toSeq\ a$：与 Sequence 的类似，基于 Array 的实现是 $W = S = 1$，由于基于 Tree 的实现 $W = |a|,\ S = lg|s|$。
- $intersection,\ union\ difference$：就是集合运算，Array 实现下 $W = |a|,\ S = 1$，Tree 下实现 $W = mlg(1 + \frac{n}{m}),\ S = lg(n)$，这与 BST 一致。
- $find,\ insert\ delete$：find 在 Array 实现下 $W = S = 1$, 其余两个在 Array 实现下 $W = |a|,\ S = 1$，而 Tree 实现下三个均为 $W = S = lg|a|$。事实上，我们可以用三个集合基本运算来实现这三个操作，当两个集合尺寸差别很大时，我们有

  $$W \in O(1 \times lg(1 + \frac{n}{1})) = O(lgn)$$

  当两个集合尺寸相近时，我们有

  $$W \in O(nlg(1 + \frac{n}{n})) = O(n)$$

- $filter\ f\ a$：Sequence 实现下 $W = |a| + \sum_{x \in a} W(f(x)),\ S = 1 + \max_{x \in a} S(f(x))$，Tree 实现下 $W = \sum_{x \in a} W(f(x)), ; S = lg|a| + \max_{x \in a} S(f(x))$。

Table 是键值对的集合，是 Key 集合和 Value 集合的直积（笛卡尔积），即每个 Key 至少出现一次。Table 的类集合运算全是基于 Key 的.

- $filter\ p\ a,\ map\ f\ a$：前者 $W = \sum_{(k \to v) \in a} W(p(k, v)),\ S = lg|a| + \max_{(k \to v) \in a} S(p(k, v))$，后者 $W = \sum_{(k \to v) \in a} W(f(v)),\ S = lg|a| + \max_{(k \to v) \in a} S(f(v))$. 从这个角度看，Table 实现方式还是基于 Tree 的。
- $union\ a\ b,\ intersect\ a\ b,\ difference\ a\ b,\ restrict\ a\ c,\ subtract\ a\ c$:

  $$W = mlg(m + n)$$

  $$S = lg(m + n)$$

  其中 *restrict* 是把原 Table 限制在给定 Key 域内的键值对；*substract* 是从原 Table 中处于给定 Key 域内的键值对删除。这两者是互易的操作。
- $find\ a\ k,\ delete\ a\ k,\ insert\ f\ a\ (k, v)$:

  $$W = S = lg|a|$$

此外，还有 Ordered Sets 和 Ordered Table，它们的操作和 Treap 十分类似，由于实现基于 Tree，因此复杂度也与 Treap 对应的操作基本一致。

2.4 算法的证明策略

我们知道，常用的数学证明方法有分析法、反证法和归纳法，而算法正确性的证明也通常基于这三种策略进行，并且要结合算法具体的实现步骤。下面就列举课程中见到的几种算法证明以获得通用的算法证明思路。这些思路在后面的实验中同样会用到。除了证明算法的正确性之外，还要证明算法是收敛的，即不论出现何种情况，算法总能结束，不会出现无穷递归或死循环之类的情况。

例子： Genome Sequencing

这个问题可以简单描述为寻找一条包含了所有子串的最小父串，即基因序列集合中所有的片段都能在父串中匹配到，而这条父串又是所有符合条件的父串中长度最短的。除了暴力求解外，还可以将其转化为 TSP 问题，即将所有片段作为结点构成一个多重有向加权图，解决问题就是寻找整个图的哈密顿路，然而这两种求解方法都是 NP 难 (Non-Deterministic Polynomial)。贪心法可以在 $W(n) \in O(nm^2), S(n) \in O(nlgm)$ 内完成，下面我们证明它的正确性，即贪心法确实返回一个所有原始片段的父串。

证明. 根据算法的步骤, 每一次迭代都会在片段集合中贪心地选取两个拥有最大重叠部分长度的串并从最大重叠部分合并它们，而迭代的终止条件是集合中只剩下一个串并且将其作为结果父串返回。由于每次子串集合中的串数量都会减少一，则最终子串集合一定会成为空集，算法的收敛性得到了保证。因此，如果返回的串不包含全部子串，则集合中必定还有没被合并的子串，这说明集合中剩下不止一个串，与迭代终止的条件矛盾。故返回的父串必定包含所有子串。而由于贪心策略只是在迭代每一步中选择局部最大重叠部分，有可能会错过全局最大重叠部分，故返回的父串不一定是最短父串，但接近最短父串。从数学期望的角度来说，返回的串长度是理论最短父串长度的 2 3.5 倍。

□

然而，贪心方法与转化为旅行商问题的方法步骤本质上是不同的。旅行商版本是将子串集合中所有子串作为图结点，有向边的权值就是出发结点的串长度与两顶点串重叠部分长度之差，而这个图的哈密顿路即为所求。显然，前者选择合并串的条件与后者选择路的条件不同：前者的选择不受当前父串的大小影响，而后者边的权值还与父串大小有关，这也解释了为什么后者不是对称有向图。因此，前者的结果是将所有重叠部分最大的串拼接在一起作为新父串，而后者的结果是将所有需要进行扩展改动最小的串拼接在一起作为新父串。在串长度不确定情况下，这两者并不是等价的。

第三章　实验 1：无重复排序

3.1　实验介绍

对给定 n 元素序列进行不重复排序，本实验使用快速排序算法求解，对三种选取主元的策略（选取首元素，选取首、中间、尾元素的中位数、随机选取）进行了比较。通过数学推导，得出结论：前两种策略受坏情况影响较大，最坏 $W \in \Theta(n^2)$，正常 $W \in \Theta(nlgn)$，而随机选取策略较稳定，$W = O(nlgn)$, $S = O(lg^2 n)$。

3.2　问题描述

给定一个 n 个元素的不重复序列，将其升序排列。在这里我们将使用快速排序进行求解。

Algorithm 1: 快速排序（伪代码）

```
Data: Sequence a
Result: Increasing Sorted Sequence a'
quicksort a =
if |a| = 0 then
    a
else
    let
        p ← pick a pivot from a
        a1 ← ⟨x ∈ a|x < p⟩
        a2 ← ⟨x ∈ a|x = p⟩
        a3 ← ⟨x ∈ a|x > p⟩
    in
        a1 + +a2 + +a3
    end
end
```

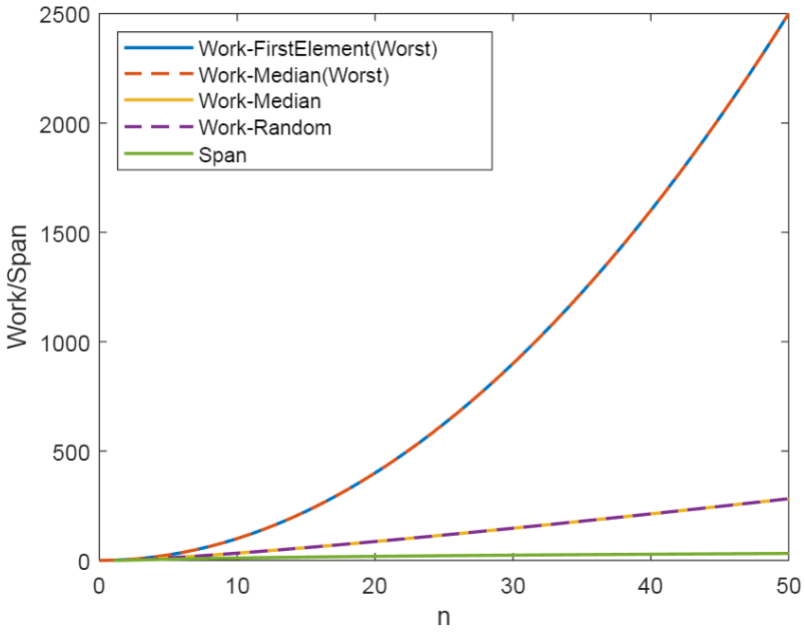

图 3.1: 三种主元选取策略复杂度比较

3.3 算法

3.3.1 算法描述：快速排序

对于这个任务首先想到的是传统快速排序，它是分治法的典型代表，每次从序列中选择一个主元（选择策略由实现决定），将主元与位于序列中间的元素互换，将小于主元的元素移动至主元左侧，大于主元的元素移动至主元右侧。递归地进行这一过程。递归的退出条件，即基本状态，是序列长度为 0.

3.3.2 算法分析

快速排序有很高的并行度。例如，相邻的调用快排函数的递归是可以并行的，使用 $filter$ 对每个序列筛选比主元大、比主元小、与主元相等的操作也可以并行。对每一次对快排函数调用不是进入基本情况，就是会产生两个递归栈。因此，可以用二叉树来描述快排的过程。
复杂度主要由比较过程，即实现中的筛选决定，根据 $filter$ 的性质，我们可以确定 $W \in \Theta(n)$，$S \in \Theta(lgn)$。整体算法的复杂度与选取主元的策略有关，这里我们分析三种主元选取策略。[图 3.1]

- 始终取首元素：这种策略受坏情况，即输入为一个基本有序序列时影响较大。会形成一个不平衡的树，深度为 n，因此总工作量是

$$W \in \Theta(n^2)$$

- 始终取首、中间、尾元素的中位数：生成一个相对平衡的树，深度为 $\Theta(lgn)$ 它能够对序列进行相

对平衡的划分，因此有

$$W \in \Theta(nlgn)$$

然而，在最坏情况，即输入一个在首、中间、尾元素恰好都是较小值或较大值时会退化为策略 1。

- 随机选择：生成一个相对平衡的树，尽管这是一个不确定算法，但通过对树高求期望我们仍然可以得到它的复杂度。

$$S(n) = S(X(n)) + O(lgn)$$

$$E(S(n)) = O(lg^2 n)$$

因此，对于随机取主元的策略，我们有

$$W \in \Theta(nlgn),\ S \in \Theta(lg^2 n)$$

事实上，如果在随机选择策略下显式地给定出每个元素的优先级，这意味着我们隐式地使用了 $Treaps$ 这种基于树的数据结构，它使得我们的快速排序算法成为一个确定性算法，下面证明在给出元素优先级时，生成的 Treap 是唯一的，即生成的排序二叉树是唯一的。

证明. 使用归纳法证明。首先，当一个 Treap 只有一个结点即根结点时，它是唯一的。接下来，假设一个 k 结点的 Treap 是唯一的，下面证明 k+1 个结点的 Treap 是唯一的。由于根结点必然是首先被选择的元素，他的优先级是最大的，因此根结点是唯一的。其次，根据快速排序的算法，根结点的左子树的元素都是小于根结点值的元素，而右子树的元素都是大于根结点值的元素，因此左右子树所包好的元素都是确定的。最后，由于左右子树都是 Treap，且大小不会超过 $k+1-1=k$，因此左右子树都是唯一的。于是 K+1 个结点的 Treap 是唯一的。 □

3.4 实验

3.4.1 样例分析：选取首元素为主元

输入：7,4,2,3,5,8,1,6. 注意，这并不是最坏情况的输入，但相当坏。

递归层	a
1	(3,4,2,5,6,1),(7),(8)
2	(2,1),(3),(4,5,6),(7),(8)
3	(1),(2),(3),(4),(5),(6),(7),(8)

表 3.1: 样例分析：选取首元素为主元

3.4.2 样例分析：选取首、中间、尾元素为主元

输入：7,4,2,3,5,8,1,6. 注意，这并不是最坏情况的输入，但相当坏。

递归层	a
1	(5,4,2,1,3),(6),(7,8)
2	(2,1),(3),(5,4),(6),(7),(8)
3	(1),(2),(3),(4),(5),(6),(7),(8)

表 3.2: 样例分析：选取首、中间、尾元素为主元

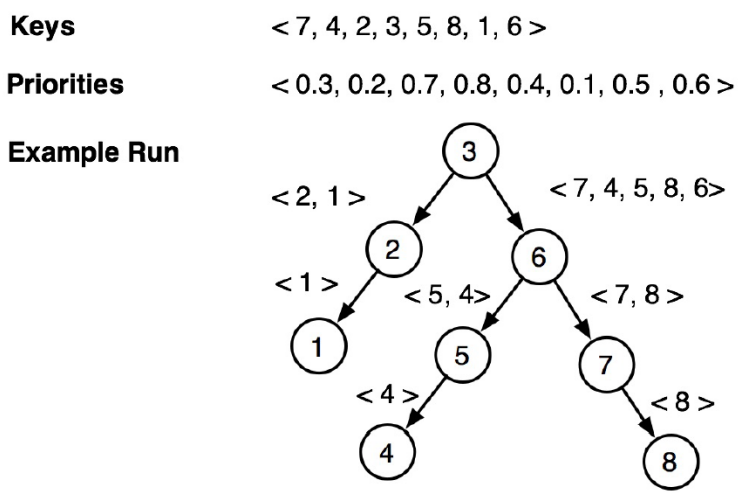

图 3.2: 随机主元选取概率图示

3.4.3 样例分析：随机选取主元

输入：7,4,2,3,5,8,1,6. 注意，为了方便分析，我们引入一个类似优先级序列概率序列来描述不同元素被选中的概率分布，如 [图 3.2] 所示。因此，我们的排序步骤可以用 [图 3.3] 描述。

递归层	a
1	(2,1),(3),(7,4,5,8,6)
2	(1),(2),(3),(4,5),(6),(7,8)
3	(1),(2),(3),(4),(5),(6),(7),(8)

表 3.3: 样例分析：随机选取主元

3.5 总结

对给定 n 元素序列进行不重复排序，本实验使用快速排序算法求解，对三种选取主元的策略（选取首元素，选取首、中间、尾元素的中位数、随机选取）进行了比较。通过数学推导，得出结论：前两种策略受坏情况影响较大，最坏 $W \in \Theta(n^2)$，正常 $W \in \Theta(nlgn)$，而随机选取策略较稳定，$W = O(nlgn)$, $S = O(lg^2 n)$。

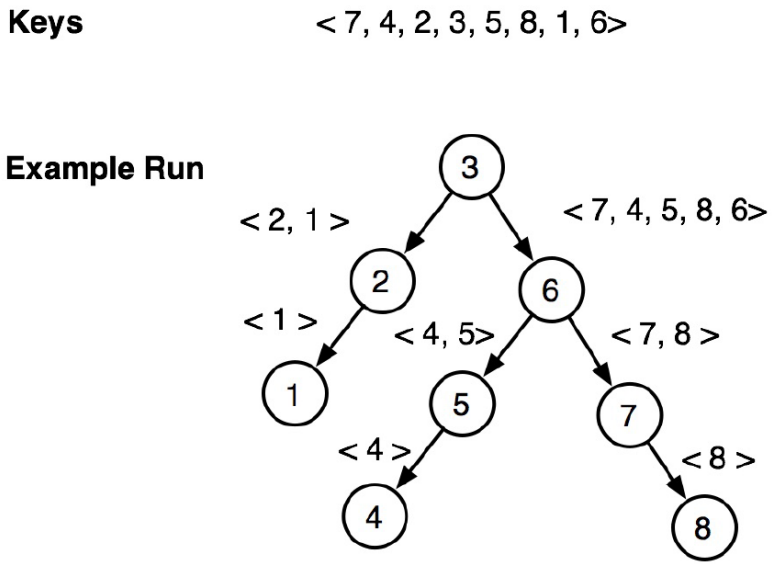

图 3.3: 图示快排算法运行过程

第四章　实验 2：最短路径

4.1　实验介绍

在有向赋权图中寻找单源最短路，本实验采用 Dijkstra 算法，该算法只能串行执行并且 $W = S = O(mlgn)$. 由于 Dijkstra 算法具有不能处理带负数权值边的图，我们随后又讨论了 Bellman-Ford 算法来弥补这个缺陷。前者 $W = S \in O(mlgn)$，后者 $W \in O(nmlgn)$，$S \in O(nlgn)$。由于实验测试数据没有带负数权值边的图，故我们仅实现了前者。

4.2　问题描述

单源最短路问题，就是给定一个有向赋权图 $G = (V, E, w)$ 和一个源结点 s，找到从源结点 s 到任意其他结点 V 的最短赋权路径。

4.3　问题分析

深入分析单源最短路问题，从子路径的性质开始。

子路径性质：任何最短路径的任意子路径是最短路径。

证明. 假设最短路径 p 有一条子路径 q 不是自己起始点之间的最短路径，那么必然存在一条 q'< q，将 q 替换为 q'，得到一条新路径 p'< p，与 p 是最短路径矛盾。故子路径性质成立。 □

下面定义 $frontier$。

结点集 X 的 $frontier$: 是与结点集为邻居且不属于 X 的结点集合，

$$i.e. \quad N^+(X) \setminus X$$

$$v \in F, x \in X, p(v) = \min_{x \in X}(\delta_G(s, x) + w(x, v))$$
$$\Rightarrow Y = V \setminus X, y \in Y, \min_{y \in Y} p(y) = \min_{y \in Y} \delta_G(s, y)$$

这种性质的直接解释就是从源结点经过 V 中结点到达其邻居 Y 中结点的最短路径就是从源结点直接到 Y 中结点的最短路径。

4.4 算法

4.4.1 **算法描述：**Dijkstra **算法**

- 从源结点开始 $d(s) = 0$,
- 计算当前结点到相邻节点最小权值 $p(v) = \min_{x \in X}(d(x) + w(x + v))$,
- 更新权值 $d(v) = p(v)$

因此，Dijkstra 算法按照非严格单调递增的最短路径长度顺序依次访问未访问结点。

从实现上来看，Dijkstra 算法与广度优先搜索有些类似。事实上，广度优先搜索可以被视为从 X 到 Y 的所有路径权值相等的 Dijkstra 算法。我们给出给予优先级队列实现的 Dijkstra 算法的伪代码。

4.4.2 **算法描述：**Bellman-Ford **算法**

然而，dijkstra 算法拥有致命缺陷，那就是无法处理带负数权值的图，当遇到这种图时，它会不断地在负数权值边上来回往复来降低全局路径长度——这是荒谬的，因此，我们可以在它失效时使用 Bellman-Ford 算法。

在 dijkstra 算法中，我们通过逐渐加入结点和边来计算最小权值。在这里不同的是，我们首先就要加入越来越多的边，并迭代地通过计算某一结点到达各个相邻结点的路径长度之和，与当前的最短路径进行比较，若短于当前最短路径则更新，直到这个结点的路径长度不再变化为止。这个过程称为 $relax$。算法对图的每条边进行 $|V| - 1$ 次处理，每一次处理对每条边进行一次松弛操作。

4.4.3 **算法分析**

在实现中，有优先级队列和基于树结构的表两种数据结构可供选择。它们的操作有细微差别，但总复杂度近似相等。令 $n = |V|$, $m = |E|$，优先级队列的主要操作是删除最小元素和将新结点插入队列，它们的工作量都是 $O(lgn)$，因此，我们有

$$W = S = O(mlgn)$$

注意，dijkstra 算法只能串行执行。

由于 Bellman-Ford 算法对于每个结点周围的所有边进行迭代可以并行执行，因此有

$$W(n, m) = O(nmlgn),\ S(n, m) = O(nlgn)$$

可以注意到，Span 与图的边数无关，正说明了对边的迭代可以并行执行这一性质。因此，在可能含有较多的环（这是 Bellman-Ford 所擅长的），即边数多于结点数的图作为输入，我们可以看到 Bellman-Ford 算法的 Span 还要优于 Dijkstra 的 [图 4.1]。

Algorithm 2: Dijkstra 算法

Data: $G = (V, E, w), source\ nodes$

Result: $Prior\ Queue\ Q$

```
1  DijstraPQ G s =
2  initialize: Q_0 ← insert(0, s)
3  let
4      dijkstra X Q =
5      if empty(Q) then
6          X
7      else
8          if v already visited then
9              dijkstra X Q'
10         else
11             let
12                 X' ← X ∪ {(v, d)}
13                 relax (Q, (u, w)) ← insert (d + w, u) to Q
14                 Q'' ← iterate relax Q'(N_G^+(v))
15             in
16                 dijkstra X Q"
17             end
18         end
19     end
20 in
21     dijkstra {} Q_0
22 end
```

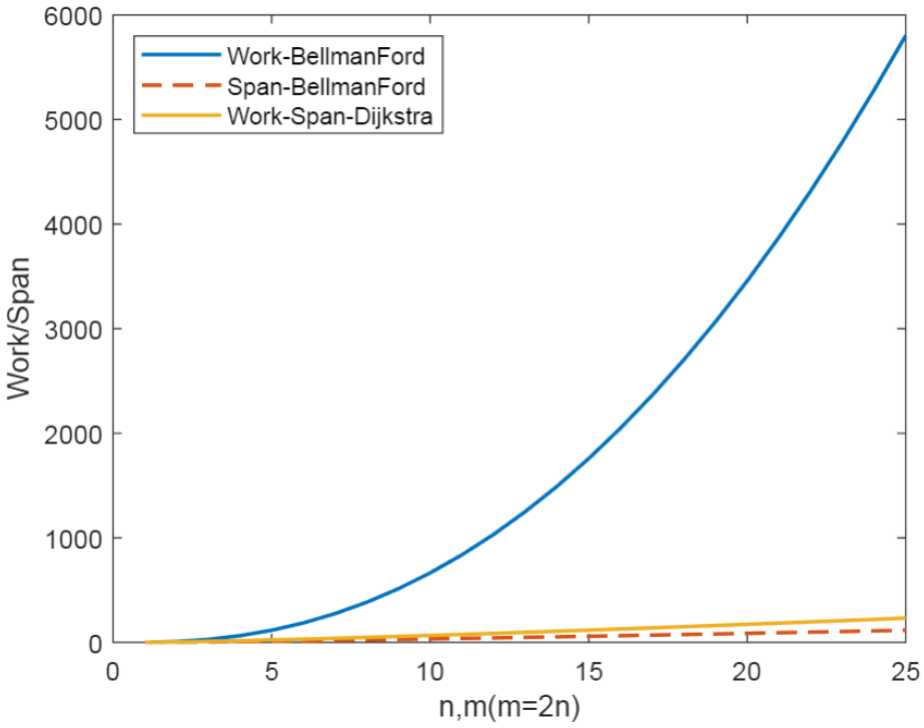

图 4.1: Dijkstra,Bellman-Ford 算法复杂度比较

4.5 实验

输入：一个表示图的临接表 [图 4.2]

$$G = (s \to (a \to 1, c \to 5), a \to (b \to 2), b \to (c \to 1, d \to 5), c \to (d \to 3), e \to (d \to 0))$$

. 将优先级队列 Q 初始化为 $Q = (s, 0)$. 算法运行过程如 [表 4.1]。

4.6 总结

在有向赋权图中寻找单源最短路，本实验采用 Dijkstra 算法，该算法只能串行执行并且 $W = S = O(mlgn)$. 由于 Dijkstra 算法具有不能处理带负数权值边的图，我们随后又讨论了 Bellman-Ford 算法来弥补这个缺陷。前者 $W = S \in O(mlgn)$，后者 $W \in O(nmlgn)$，$S \in O(nlgn)$。由于实验测试数据没有带负数权值边的图，故我们仅实现了前者。

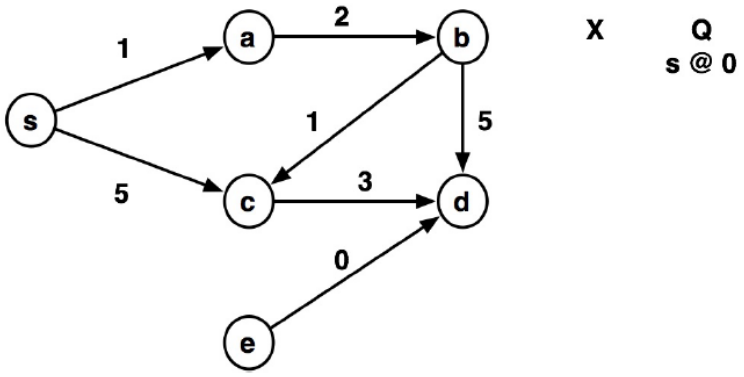

图 4.2: 输入示例

X	Q
	(s,0)
(s,0)	(a,1)
	(c,5)
(s,0)	(b,3)
(a,1)	(c,5)
(s,0)	(c,4)
(a,1)	(c,5)
(b,3)	(d,8)
(s,0)	(c,5)
(a,1)	(a,6)
(b,3)	(d,7)
(c,4)	(d,8)
(s,0)	(d,7)
(a,1)	(d,8)
(b,3)	
(c,4)	
(s,0)	(d,8)
(a,1)	
(b,3)	
(c,4)	
(d,7)	
(s,0)	
(a,1)	
(b,3)	
(c,4)	
(d,7)	

表 4.1: Dijkstra 算法测试样例

第五章　实验 3：最大括号长度

5.1　实验介绍

求闭合串中最大括号长度，可以使用暴力方法、非递归栈结构和递归分治策略三种方法，复杂度分别为 $W \in O(n^2), W \in O(n), S \in O(nlgn)$。

5.2　问题描述

一个串是闭合的，当且仅当它只包含"(","）" 并满足以下条件的一个或多个：

- 它由两个闭合串连接而成。
- 它是一个单一的闭合串并由一对括号包围。
- 它含有一对匹配的括号，其中没有任何其它元素。

我们给出“括号长度”的定义，就是在一对匹配的括号之间的字符个数，不包括括号。而最大括号长度就是所有匹配括号串长度中最大的长度。

5.3　算法

5.3.1　算法描述

最基础的是在每一次迭代中枚举所有的括号长度并取其中最大值的暴力求解方法，也可以通过串行的栈数据结构来求解这个问题，也可以递归地使用分治策略完成。这里我们采用的是串行非递归算法，要借助一个类似栈的数据结构，实际上是维护了一个先进后出的序列。由于题目声明给定的串是闭合串，因此我们只需要用 1，-1 代表左右括号并以前缀和的值作为栈的状态即可，无需判断此时是否有不匹配括号出现。具体状态转移规则如下：

- 出现一个左括号则入栈，括号距离不变，计数加一。
- 出现一个右括号，将括号距离更新为该右括号与栈顶左括号距离，并将左括号出栈，同时计数加一。

Algorithm 3: 最大括号长度

```
Data: Close Sequence a
Result: Maximum Parenthesis Distance d
maxDistance a =
let
    count(s,d,x,c) =
    switch (s,d,x,c) do
        case (Some n, _,'(', _) do
            Some(n + 1)
            c ← c + 1
        end
        case (Some n, _,')', _) do
            Some(n − 1)
            if n = 0 then
                d ← c − 1
                c ← 0
            end
            if c > d then
                d ← c − 1
            end
        end
    end
in
    (iterate count (Some n, d, 0) a) = (Some n, 0, 0)
end
```

5.3.2 算法分析

暴力求解法是每次迭代中枚举所有的情况，因此除了迭代 n 次外，在每一次中也有 $W = O(n)$，因此暴力求解法 $W \in O(n^2)$。

串行操作是迭代进行，根据输入串长度迭代 n 次，每一次都能在常数时间内完成，因此 $W = S = O(n)$。

此外，如果用递归的分治策略，复杂度满足

$$W(n) = 2W(\frac{n}{2}) + W(lgn) + O(1)$$

$$S(n) = 2S(\frac{n}{2}) + O(1)$$

求解得到

$$W(n) \in O(nlgn),\ S(n) \in O(lg^2 n)$$

两者的渐进对比如 [图 5.1]。

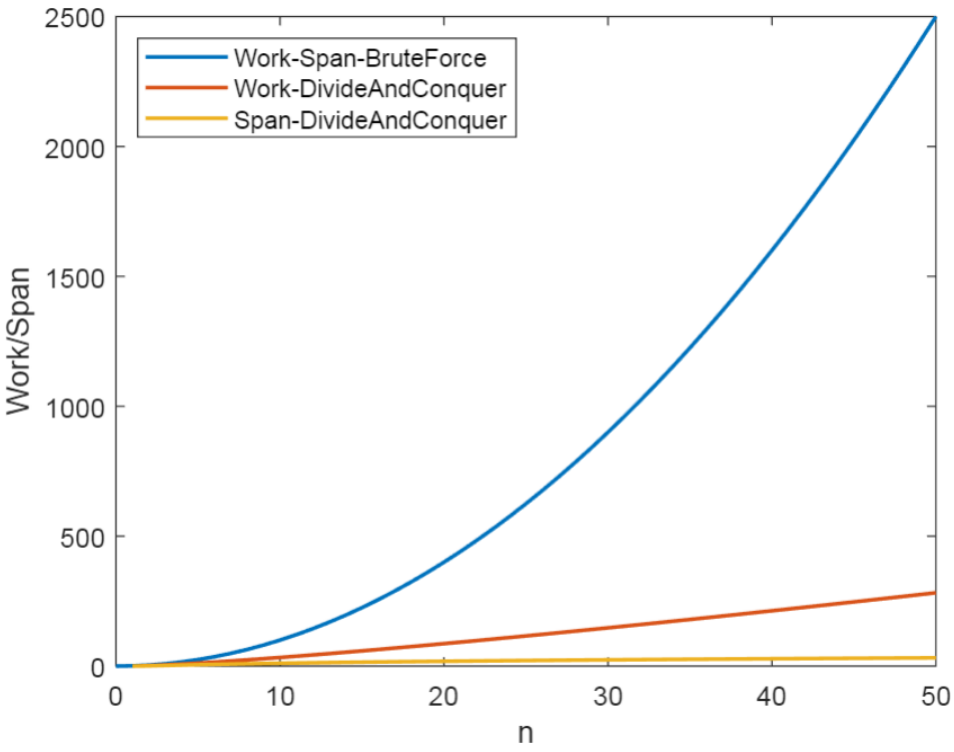

图 5.1: 暴力求解与分治算法复杂度比较

5.4 实验

输入：(()(()))

s	x	c	d
0	0	0	0
1	1	0	0
2	1	1	0
1	-1	2	0
2	1	3	0
3	1	4	0
2	-1	5	0
1	-1	6	0
0	-1	7	6

表 5.1: 样例分析

5.5 总结

求闭合串中最大括号长度，可以使用暴力方法、非递归栈结构和递归分治策略三种方法，复杂度分别为 $W \in O(n^2), W \in O(n), S \in O(nlgn)$。

第六章　实验 4：天际线

6.1　实验介绍

天际线问题，就是找到对于每个给定的建筑物的左右坐标点对应位置的楼房最大高度，可以用线段树法、扫描线法和分治法。本实验中讨论了扫描线法和分治法，前者复杂度是 $W = S \in O(nlgn)$，后者复杂度是 $W \in O(nlgn), S\ O(lg^2 n)$。

6.2　问题描述

给定非空的建筑集合 $B = \{b_1, \ldots, b_n\}$，每个建筑由三元组 $b_i = (l_i, h_i, r_i)$ 表示，分别代表其左坐标、高度和右坐标。我们的目标是找到一个点集 $S = \{p_1, \ldots, p_2 n\}$，满足

$$p_j = (x_j, y_j) = (x, \max(h : (l, h, r \in B))) : x \in \bigcup_{(l,h,r)\in B} \{l, r\}$$

一言以蔽之，就是找到对于每个给定的建筑物的左右坐标点对应位置的楼房最大高度 [图 6.1]。

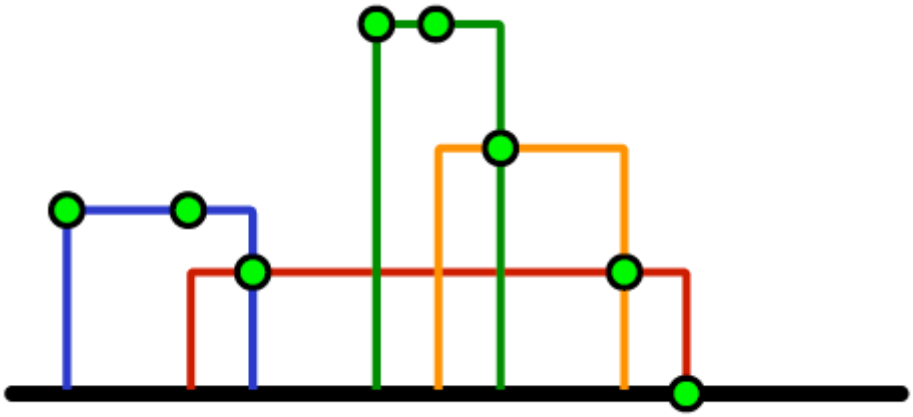

图 6.1: 题目图示

6.3 算法

6.3.1 算法描述

求解这个问题，可以使用线段树法、扫描线法和可并行化分治法。首先将高度离散化，即 $H = \{h_0 = 0, h_1 = 1 \ldots h_{n-1} = n-1\}$（按照大小映射到 0，1，2⋯⋯n），然后将楼房的左边界和右边界一起排序。

同时维护一个串，每当遇到一个边界则让该串从 0 到离散化之后的边界高度的元素 +1（左边界）或-1（右边界），并查询修改后该串最大的不是 0 的元素的位置对应的高度和修改前该串最大的不是 0 的元素的位置对应的高度比较，如果不一样则输出。这就是扫描线法。

另外一个思路就是分治策略下的天际线。这里要用到 $Treaps$ 这种数据结构。我们将每个建筑物的表示由原先的三元组 (l, h, r) 变为 $((l, h), (r, h))$，即左上角和右上角顶点。为区分左顶点和右顶点，我们将左顶点坐标赋值为原值的相反数。构造一个 $Treap$，对建筑物集合进行遍历。

- 如果遇到左顶点则将其插入。通过将这个堆维护为大顶堆，我们保证了处在根结点的元素始终是当前最高的建筑物。
- 如果遇到右顶点，则在堆中搜索到对应的左顶点，将其删除，这意味着这个建筑物已经结束。
- 对于结果的记录，只需要记录每次根结点元素变化时对应的横坐标和高度即可。

下面给出分治策略下天际线问题的伪代码。

6.3.2 算法分析

对于不引入线段树的简单线扫描算法，由于每次向右重新查询最大高度. 使用二分查找需要 $W \in O(lgn)$，而顺序遍历所有建筑物元素需要 $W \in O(n)$，因此串行扫描线算法有

$$W = S \in O(nlgn)$$

对于分治策略，由于我们采用 $reduce$ 进行分治，在 Array 实现下，有

$$W = 1 + \sum W(skyLine)$$
$$S = lg|s|max(S(skyLine))$$

而 skyLine 函数每次调用主要开销在对 Treap 的 delete 和 insert 操作上，它们的复杂度满足

$$W \in O(lgn)$$

$$S \in O(lgn)$$

因此，最终我们有

$$W \in O(nlgn)$$

$$S \in O(lg^2 n)$$

扫描线算法与分治算法的复杂度对比见 [图 6.2]。

Algorithm 4: 天际线问题

Data: $B = \{b_1, \ldots, b_n\}$

Result: $p_j = (x_j, y_j)$

```
initialize:B' ← {(( b1.l, b1.h), (b1.r, b1.h)), ... }
skyLine B' T p =
let
    switch second B' do
        case second B' < 0 do
            if second B' < T.k then
                insert T second B'
                p ← p append ( first B', second B')
            else
                insert T second B'
            end
        end
        case second B' < 0 do
            delete T first B'
        end
    end
in
    reduce skyLine B' (_, _, _) ()
end
```

6.4 实验

输入:$B = \{(1,2,3),(2,1,4),(2,3,5),(3,1,6)\}$ 预处理:$B' = \{((1,2),(3,2)),((2,1),(4,1)),((2,3),(5,3)),((3,1),$

递归层	k	p
1	2,1,3,1	((1,2)),((2,1)),((2,3)),((3,1))
2	2,3	((1,2),(3,1)),((2,3),(5,1))
3	3	((1,2),(2,3),(5,1))

表 6.1: 分治策略天际线运行过程

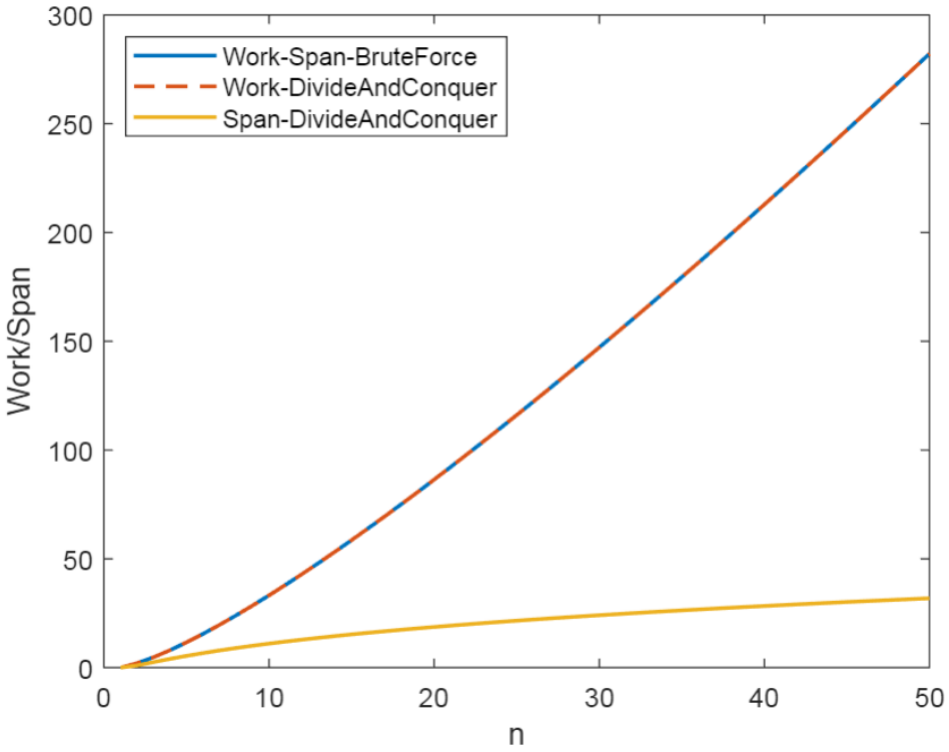

图 6.2: 扫描线算法与分治算法复杂度比较

6.5 总结

天际线问题，就是找到对于每个给定的建筑物的左右坐标点对应位置的楼房最大高度，可以用线段树法、扫描线法和分治法。本实验中讨论了扫描线法和分治法，前者复杂度是 $W = S \in O(nlgn)$，后者复杂度是 $W \in O(nlgn), S\ O(lg^2n)$。

第七章　实验 5：括号匹配

7.1　实验介绍

判断一个字符串序列是否是括号匹配的，本实验中我们使用了只能串行执行的 Iterate 方法和可并行执行的 Scan 方法来实现括号匹配算法，并推导出前者 $W = S = O(n)$，后者 $W = O(n)$，$S = O(lgn)$.

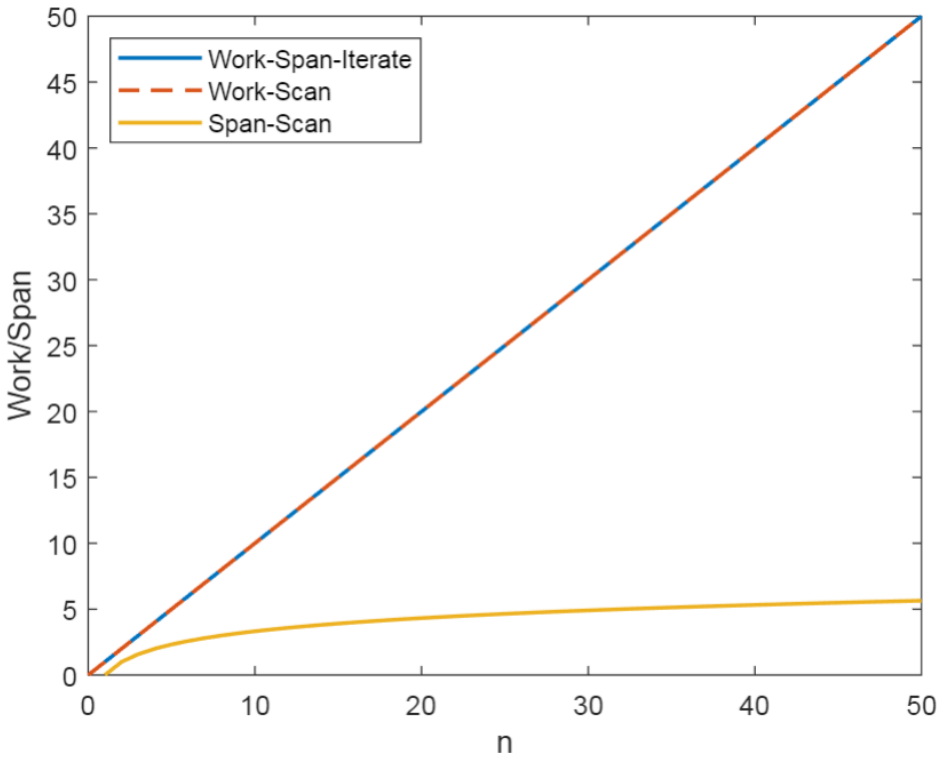

图 7.1: Iterate 与 Scan 复杂度比较

7.2 问题描述

给定一个字符串序列，判断它是否是括号匹配的，即所有的括号都是闭合的。关于闭合串的定义我们在实验 3 已经给出。

Algorithm 5: 括号匹配

```
Data: Sequence a
Result: Sum of matched parenthesises x
matchParens a =
let
    count(s,x) =
    switch (s,x) do
        case (None, _) do
            None
        end
        case (Some n,'(') do
            Some(n + 1)
        end
        case (Some n,')') do
            if n = 0 then
                None
            else
                Some(n − 1)
            end
        end
    end
in
    (iterate count (Some, 0) a) = (Some, 0)
end
```

7.3 算法

7.3.1 算法描述

首先描述一种基于迭代的匹配算法，使用状态变量构成一个二元组进行状态判定（实际上这个实验就是实验 2 的简化版）。在算法的实现中，我们隐式地构造了一个类似栈的先进后出的数据结构。

- 对于错误状态，返回错误状态；

- 对于正确状态，且接受左括号，则左括号入栈；
- 对于正确状态，且接受右括号：若栈为空，则返回错误状态；反之左括号出栈完成一次匹配。

这种基于迭代的匹配算法并行度非常低，将伪代码第 20 行的迭代语句换成

$$(scan\ count\ (Some, 0)\ a) = (Some, 0)$$

我们就利用 scan 的对所有前缀子序列进行并行 reduce 操作的特点完成了并行化。在此基础上，将出入栈策略改为：接受左括号返回值 1，接受右括号返回值-1，判断前缀和是否为 0，若前缀和为 0 即为前缀子串是括号匹配的。需要注意的是，*scan* 操作的函数必须满足结合律，即

$$(a\ op\ b)\ op\ c = a\ op\ (b\ op\ c)$$

即操作符两边的数据类型是一致的，否则就失去了意义。因此，我们把前缀和作为操作符两侧的数据。

7.3.2 算法分析

对于 iterate 版本，长度为 n 的序列要进行 n 次迭代，而每一次迭代中要对串进行遍历匹配，因此有

$$W = S = O(n)$$

对于 scan 版本，scan 所有前缀子串可以并行执行 $S = O(1)$，而 reduce 则有 $S = O(lgn)$，因此我们有

$$W = O(n),\ S = O(lgn)$$

两者的渐进比较如 [图 7.1]。

7.4 实验

输入样例：(()))())。这里分别使用 Iterate 方法 [表 7.1] 和 Scan 方法 [表 7.2] 进行模拟测试。

7.5 总结

判断一个字符串序列是否是括号匹配的，本实验中我们使用了只能串行执行的 Iterate 方法和可并行执行的 Scan 方法来实现括号匹配算法，并推导出前者 $W = S = O(n)$，后者 $W = O(n),\ S = O(lgn)$.

Iteration	s	x
0	Some	0
1	Some	1
2	Some	2
3	Some	1
4	Some	0
5	None	-
6	None	-
7	None	-
8	None	-

表 7.1: Iterate 方法测试样例

递归层	s’	x
0	Some	0,0
1	Some	1,1
2	(Some,Some)	(1,2),2
3	(Some,Some,Some)	(1,2,1),1
4	(Some,Some,Some,Some)	(1,2,1,0),0
5	(Some,Some,Some,Some,None)	(1,2,1,0,-1),-1
6	(Some,Some,Some,Some,None,None)	(1,2,1,0,-1,-),-
7	(Some,Some,Some,Some,None,None,None)	(1,2,1,0,-1,-,-),-
8	(Some,Some,Some,Some,None,None,None,None)	(1,2,1,0,-1,-,-,-),-

表 7.2: Scan 方法测试样例

第八章　实验 6：高精度运算

8.1　实验介绍

计算两个大整数的和、差、积，我们使用传统串行的竖式加法、减法和乘法，讨论了一种基于分治策略的可并行乘法，推导出加减法复杂度为 $O(n)$，串行乘法复杂度为 $O(n^2)$，并行复杂度为 $S \in O(lg^2 n)$。

8.2　问题描述

给定两个任意精度整数，满足 $a \geq b$，分别求出 $a+b, a-b, a \times b$ 的值。

8.3　算法

8.3.1　算法描述

由于计算机中不存在任意大精度的整数数据类型，因此必须将输入的数据转化为一个数字串序列，再通过竖式整数运算，即对每一对应位进行运算操作并进位的方式求解。

对于加法运算，每一位完成对应加法之后乘十进位，而对于乘法运算，每一位完成对应乘法后乘十进位，第二乘数的每一位乘法结果乘十相应位数次方相加。通过这种性质我们可以知道，高精度浮点运算实际上是不满足交换律的，两个乘数交换导致的结果可能有细微的差别。相应在实现中需要注意的

Algorithm 6: 高精度加法

Data: $a \geq b, a = (a_{n-1} \dots a_0)_{10}, b = (b_{n-1} \dots b_0)_{10}$

Result: s_n

```
1 c ← 0
2 for j ← 0 to n − 1 do
3 |   d ← (a_j + b_j + c) div 10
4 |   s_j ← a_j + b_j + c − 10d
5 |   c ← d
6 end
7 s_n ← c
```

Algorithm 7: 高精度乘法

Data: $a \geq b, a = (a_{n-1} \dots a_0)_{10}, b = (b_{n-1} \dots b_0)_{10}$

Result: $p = ab$

```
for j ← 0 to n − 1 do
    if b_j ≠ 0 then
        c_j ← a × b_j
    else
        c_j ← 0
    end
end
{c_0, c_1, ..., c_{n−1}} are the partial products
p ← 0
for j ← 0 to n − 1 do
    p ← p + c_j × 10^j
end
```

是，要小心前导 0，对操作数首先要清一次前导 0，减法的结果当然要清一次，关键是乘法的结果也要清前导零，防止因为 0 乘导致结果为 0 的情况。

8.3.2 算法分析

由于加法只需要对对应位进行运算操作，相当于对至多 n 个数对进行操作，故 $W \in O(n)$，加法我们只实现了串行操作。

乘法的串行操作中，由于每一位都要与另一个乘数的所有位分别进行操作，故 $W \in O(n^2)$.

乘法有可能实现并行操作吗? 答案是肯定的，这里介绍一种基于分治算法的大数乘法，尽管由于时间紧张我并没有在实验中真正地实现它。其基本思想是将两个乘数分成最重要的一半和最不重要的一半，即

$$A = p2^{n/2} + q,\ B = r2^{n/2} + s$$

则两数乘积就是

$$AB = pr2^n + (ps + rq)2^{n/2} + qs$$

在这样的分治策略下，我们可以实现乘法的并行，其复杂度满足

$$S = O(lg^2 n)$$

8.4 实验

8.4.1 样例分析

输入：a=4152,b=1341. 运行过程如 [表 8.1]。

j	c_j	p
0	4152	4152
1	16608	170232
2	12456	1415832
3	4152	5567832

表 8.1: 乘法样例分析

8.4.2 边界条件处理

在实现中需要注意的是，要小心前导 0，对操作数首先要清一次前导 0，减法的结果当然要清一次，关键是乘法的结果也要清前导零，防止因为 0 乘导致结果为 0 的情况。

8.5 总结

计算两个大整数的和、差、积，我们使用传统串行的竖式加法、减法和乘法，讨论了一种基于分治策略的可并行乘法，推导出加减法复杂度为 $O(n)$，串行乘法复杂度为 $O(n^2)$，并行复杂度为 $S \in O(lg^2 n)$。

第九章　实验 7：割点与割边的判定

9.1　实验介绍

给定一个无向无权连通图，求出图中所有割点和割边的数目，这里使用时间戳和 Tarjan 算法进行深度优先搜索实现，最优实现方式的复杂度为 $W = S \in O(m + n)$。

9.2　问题描述

要描述这个问题，首先要介绍割点和割边的概念。

- 割点：一个连通图中，如果删除一个顶点及其相连的边后，图中的连通分量数量增加，则称这个顶点是图的割点。
- 割边：一个连通图中，如果删除其中一条边，图中的连通分量数量增加，则称这条边为图的割边。

题目就是给定一个无向无权连通图，求出图中所有割点和割边的数目。

9.3　算法

9.3.1　算法描述：割点的判定

思路是比较确定的，使用深度优先搜索确定出一棵 DFS 树，使用 Tarjan 算法完成判定。在 DFS 树中，割点有如下性质。

- 若 v 是根结点，且 v 至少有两个儿子，则删除 v 后它的子树均不连通，v 是割点。
- 若 v 不是根结点，且 v 至少有一个儿子 u，从 u 及其后代出发没有指向 v 的祖先结点的边，则删除 v 后 u 为根结点的子树与祖先所在的树不连通，v 是割点。

首先应用的技术是时间戳技术，它能够帮助我们记录深度优先搜索访问和回溯的顺序 [图 9.1]。每个时间戳有两个分量，一个记录了首次访问到这个结点的时间（即这个结点第几个被访问到的结点），另一个记录了最近一次访问访问这个结点的时间，(T_V, T_F)。通过这种形式的时间戳，我们可以判断一个无向无权连通图中的非 DFS 边 $e = (u, v)$ 是 $cross, backward, forward$

- cross:$T_V(u) > T_V(v)$ and $T_F(u) > T_F(v)$

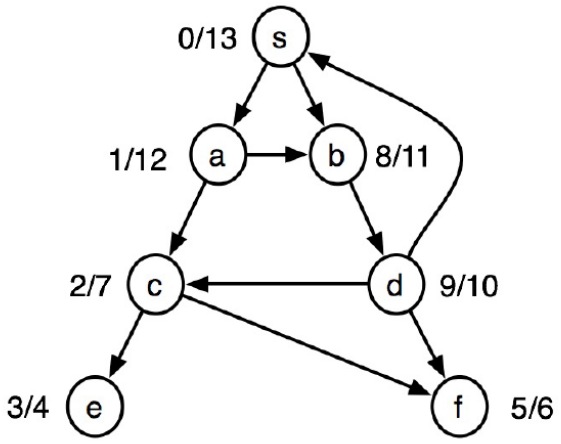

图 9.1: 时间戳

- backward: $T_V(u) > T_V(v)$ and $T_F(u) < T_F(v)$
- forward: $T_V(u) < T_V(v)$ and $T_F(u) > T_F(v)$

在这里，我们要对时间戳进行调整，它的意义是对于结点 v_i，$num(i)$ 表示首次访问时间，$low(i)$ 表示 v_i 不经过其 DFS 树对应的父结点所能够到达的时间戳最小的结点，即该点在它所在的强联通子图所在的 DFS 搜索子树中的 $num(i)$。在同一个强连通分量中，任意两结点 u,v 均满足

$$low(u) < num(v)$$

由此，判定割点的算法可以描述为

- 若 v 是 DFS 树的根结点，判断其度数，若 $deg^+ > 1$ 则是割点。
- 若 u 不是根结点，也不是根结点的直接子结点，且满足

  $$low(u) \geq num(P(u))$$

  则 u 的父亲 $v = P(u)$ 是割点。

我们的遍历过程就是

- v_i 没有被访问，打上时间戳，继续 DFS。
- v_i 已经被访问，更新 $low(i)$，使得

  $$low(i) = \min\{num(j) | v_i \ is \ connective \ to \ v_j\}$$

Algorithm 8: 判定割点

Data: $G = ((S, X), v)$

Result: P

DFSpoint G P = if $v \in X$ then
    $revisit\ S\ v\ X$
    $low(v) = \min\{num(j) | v_i\ is\ connective\ to\ v_j\}$
    if $low(v) \geq num(p(v))$ then
        $P \leftarrow P + 1$
    end
    if $v\ is\ root$ and $deg^+(v) > 1$ then
        $P \leftarrow P + 1$
    end
else
    let
        $S' \leftarrow visit\ S\ v$
        $X' \leftarrow X \cup v$
        $(S'', X'' \leftarrow iterate\ DFSpoint\ G((S'', X''), N_G^+(v))$
    in
        $finish\ S''\ v\ X''$
    end
end

9.3.2 算法描述：割边的判定

回顾割点的判定条件，核心为

$$low(u) \geq num(v)$$

对于割边，严格大于的条件显然是满足割边定义的，然而，两者相等则表明去掉连接 u, v 的边被删除后两者可以不经过该边而互通，连通度不变，因此判定割边的条件是严格的

$$low(u) > num(v)$$

此外，由于 DFS 树的根结点只有一个，割边不可能存在于那里，故省去了判断根结点度数的操作。

9.3.3 算法分析

判断割点和割边的算法十分类似，故我们把两者合二为一进行分析。按照惯例，令 $n = |V|, m = |E|$，对图进行完整 DFS 要调用 n 次 $visit$ 和 $finish$，调用 m 次 $revisit$，总共调用 $n + m$ 次 $DFSpoint$ 或 $DFSbridge$。

Algorithm 9: 判定割边

Data: $G = ((S, X), v)$

Result: B

```
1  DFSbridge G B = if v ∈ X then
2      revisit S v X
3      low(v) = min{num(j)|v_i is connective to v_j}
4      if low(v) > num(p(v)) then
5          B ← B + 1
6      end
7  else
8      let
9          S' ← visit S v
10         X' ← X ∪ v
11         (S'', X'' ← iterate DFSbridge G((S'', X''), N_G^+(v))
12     in
13         finish S'' v X''
14     end
15 end
```

证明. 由于每个顶点都在 X 中，被首次访问时都要调用 $visit$ 和 $finish$，而每次 DFS 被调而没有新顶点被访问时调用 $revisit$，这一共是 $n + m - n = m$ 次，这是因为每个顶点恰好被访问一次，它的出度边也被遍历一次，即调用一次 DFS，另外，访问 n 个顶点直接调用了 n 次 DFS，故总数为 $n + m$。 □

对于基于树结构和邻接表表示法的实现，每次调用 DFS 开销为 $O(lgn)$；对于基于暂时顺序结构和邻接序列表示法的实现，每次调用 DFS 开销为 $O(1)$。由于 DFS 的可并行度非常低，因此有

$$W = S \in O(m + n)$$

9.4 实验

9.4.1 测试样例

这里我们按照最优实现方式的邻接序列表示法来模拟。

输入：((1,2),(0,2),(0,1,3),(2,4,5),3,3)

v	X	P	B
2	()	0	0
0	(2)	1	0
1	(0,2)	1	0
3	(0,1,2)	1	0
4	(0,1,2,3)	2	1
5	(0,1,2,3,4)	2	2
3	(0,1,2,3,4,5)	2	3

表 9.1: 测试样例

9.4.2 实现细节

由于 SML 语言的特性，本实验有些实现细节需要注意。例如，为了达到理想效率，无法使用 Array2 二维数组实现临接表，而是用以 List 作为元素的一维 Array 构造邻接序列。此外，使用大小为 1 的 Array 模拟时间戳。

9.5 总结

给定一个无向无权连通图，求出图中所有割点和割边的数目，这里使用时间戳和 Tarjan 算法进行深度优先搜索实现，最优实现方式的复杂度为 $W = S \in O(m + n)$。

第十章　实验 8：静态区间查询

10.1　实验介绍

区间最值查询问题，使用在线处理方法 Sparse Table 算法，预处理开销为 $O(nlgn)$，查询开销为 $O(1)$。

10.2　问题描述

区间最值查询，给定个数列和一系列查询指令，每次查找出要求区间内最大值。

10.3　算法

10.3.1　算法描述

如果每次查询时都进行搜索，本算法可以在 $O(n)$ 内完成，但在查询量和数据量都很大的情况下就不适合了。因此，我们引进基于动态规划的 Sparse Table 算法。这个算法是在线的，当用户输入一个值后用较长时间进行预处理，随着信息的充足，可以直接在 $O(1)$ 内完成查询。

接下来描述动态规划，A_i 表示查询序列，$F(i,j)$ 表示从第 i 个数起连续 2^j 个数的最大值，状态转移方程为

$$F(i,j) = \max(F(i,j-1), F(i+2^{j-1}, j-1))$$

查询时，令 $k = \log(j-i+1)$，则

$$RMQ = \max(F(i,k), F(j-2^k+1, k))$$

10.3.2　算法分析

预处理过程类似二分查找，每个元素的预处理开销为 $O(lgn)$，故预处理总开销为 $O(nlgn)$，而查询开销为 $O(1)$。

Algorithm 10: RMQ

```
Data: A
Result: RMQ
init A max(i, j) =
let
    (_, 0) ← A
    (i, j) ← iterate init max(F(i, j − 1), F(i + 2^(j−1), j − 1))
in
    iterate init A (0, 0)
end
RMQ k =
let
    k ← max(F(i, k), F(j − 2^k + 1, k))
in
    iterate RMQ 0
end
```

10.4 实验

预处理：输入：3,2,4,5,6,8,1,2,9,7. ST 算法在实验样例上相对独立，每次处理过程大同小异，故这里不模拟完整序列，而是针对一个预处理和一个查询举例子。

$$F(1,0) = \max(3) = 3$$

$$F(1,1) = \max(3,2) = 3$$

$$F(1,2) = \max(3,2,4,5) = 5$$

$$F(1,3) = \max(3,2,4,5,6,8,1,2) = 8$$

$$\dots$$

查询：输入：1,2,3,4,5.

$$k = \log(5-1+1) = 2$$

$$result \max(F(1,2), F(2,2)) = 6$$

10.5 总结

区间最值查询问题，使用在线处理方法 Sparse Table 算法，预处理开销为 $O(nlgn)$，查询开销为 $O(1)$。

第十一章　实验 9：素性判定

11.1　实验介绍

素性测试实验，我使用 $Miller-Rabin\ Primalty\ Test$ 进行素性测试，使用线性筛法进行给定序列的素数筛选。前者的复杂度可以控制在 $O(k)$，后者的复杂度为 $W=O(nlgn), S=O(lgn)$。

11.2　问题描述

素性测试，就是给定一个整数 N，判断它是否是素数，即判断它是否是一个只能被自身和 1 整除的数。这个问题与筛选出给定整数序列中所有素数是等价的。前者秩序对一个单一元素进行素性判定，后者则是要对序列中所有的元素进行素性判定。

11.3　问题分析

素数的判断可以通过暴力求解完成，只需要枚举从 $2\ldots\lceil\sqrt{n}\rceil$ 的所有情况分别对待测数取模即可，若所有模都不为零，则是一个质数，否则就是一个合数。复杂度为 $W\in O(\sqrt{n}), S\in O(lgn)$. 尽管在素性测试中暴力求解算法复杂度并不太可能引起指数爆炸，然而在筛选素数问题，复杂度就会达到 $W\in O(n^{1.5}), S\in O(lgn)$，这对于较大的问题规模就会引起指数爆炸。

因此，这个实验中我将使用两种算法：$Miller-Rabin\ Primality\ Test$ 用于素性测试，$Prime\ Sieve$ 用于素数筛选。

11.4　算法

11.4.1　**算法描述**: $Miller-Rabin\ Primality\ Test$

算法的核心是费马小定理。

费马小定理

if $gcd(a,p)=1$, $p\in Prime$, then $a^{p-1}\equiv 1\ mod\ p$. 由此反推，如果上述关系不成立，则 a 一定是合数；反之，a 有一半的概率是素数，将进入下一次判断。这便是 M-R 素性测试的基本思想。

Algorithm 11: Miller-Rabin Primality Test

Data: $n > 2$, Parameter k which determines times of re-test

Result: $Boolean\ True/False$

```
initialization: write n-1 as 2^s × d with d odd by factorizing powers of 2 from n-1;
for i ← 1 to k do
    a ←random select from [2, n − 1];
    x ← a^(n−1) mod n;
    if x = 1 of x = n − 1 then
        do next loop;
    end
    for r ← 1 to s − 1 do
        x ← x^2 mod n;
        if x = 1 then
            return False;
        end
        if x = n − 1 then
            do next loop;
        end
        return False;
    end
    return True;
end
```

11.4.2 算法描述：素数筛

素数筛也称为埃拉托尼色筛，其基本思想就是构造一个素数表，其中的素数都是一些较小的数，针对给定的测试区间，依次迭代地筛选掉表中素数的倍数（因为素数的倍数一定是合数）。

Algorithm 12: Prime Sieve

Data: Sequence of numbers of length n

Result: Sequence of primes *primes*

```
1  primeSieve n =
2  let
3      cs ← ⟨i * j : 2 ≤ i ≤ ⌊√n⌋, 2 ≤ j ≤ n/i⟩
4      sieve ← ⟨(x, false) : x ∈ cs⟩
5      all ← ⟨true : 0 ≤ i < n⟩
6      isPrime ← inject sieve all
7      primes ← ⟨i : 2 ≤ i < n|isPrime[i] = true⟩
8  in
9      primes
10 end
```

11.4.3 算法分析

M-R 素性测试并不是确定性算法，而是随机算法。通过 M-R 素性测试的合数是伪素数和强伪素数(卡米切尔数)。当二次判定的重复次数达到 100 时（即 $k = 100$），出现错误的概率为 $(\frac{1}{2})^5 100$.

M-R 素性测试适合很大的待测数，实验中的数据最大数的有 41 位，对于暴力方法可能会超时。尽管 M-R 素性测试由于其前后结果的依赖性无法实现并行算法，但对于 k 次预定义循环，基本可以有

$$W \in O(k).$$

筛法的复杂度分析类似寻找所有子串。首先，生成范围内的合数串需要 $W = m, S = O(lgn)$，这是因为这一步需要 $Tabulate$。接下来，构造筛需要 m 的线性 Work，可并行。接下来的 $inject$ 同样需要 m 的线性 work，可并行。最后计算素数需要线性 Work 和 $O(lgn)$ 的 Span. 于是，总的复杂度为

$$W \in O(nlgn)$$

$$S \in O(lgn)$$

这样的结果明显好于暴力求解 [图 11.1]。线性筛法优于暴力求解法的原因就在于暴力求解法的工作步骤存在大量的重复，当对某一个元素用 m 取模时，我们实际上相当于同时把 $2m, 3m, \ldots$ 即 m 的倍数也作为除数，而暴力求解法中这些倍数又被重复检查，因此造成大量的浪费。线性筛法则通过删除掉这些倍数减少了冗余。

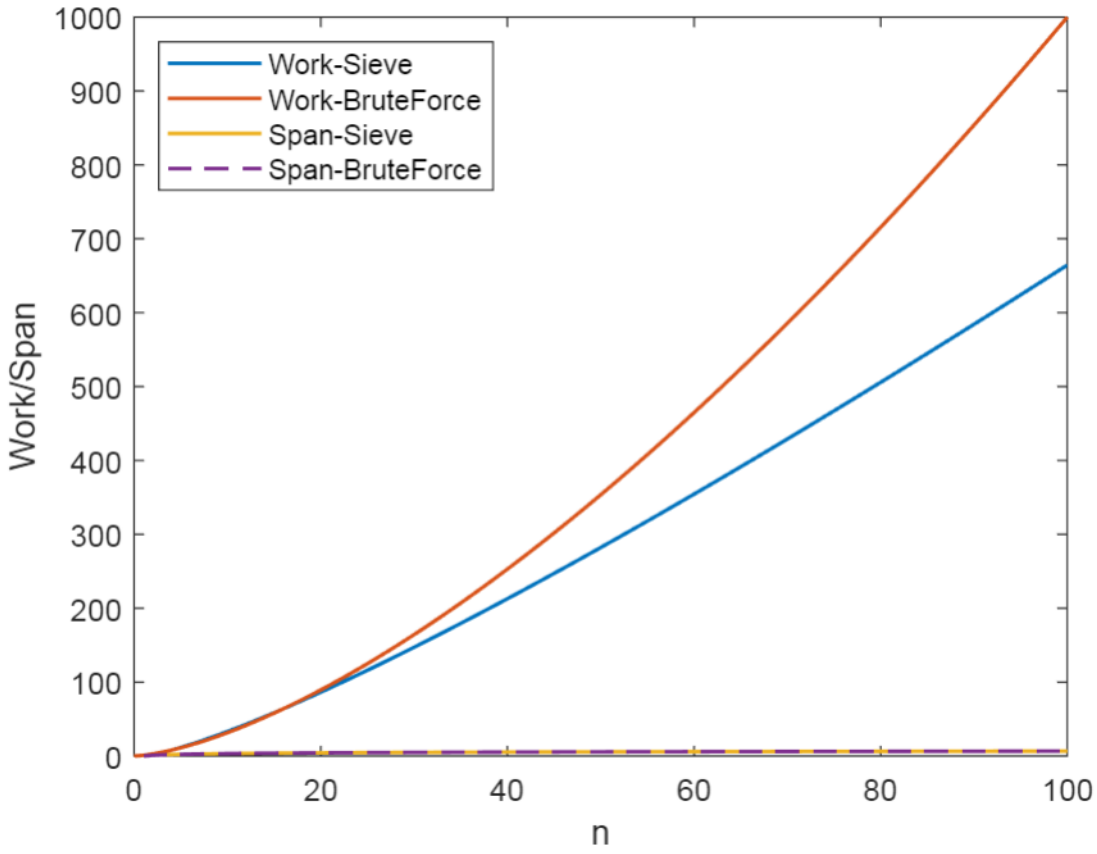

图 11.1: 暴力求解与筛法渐进时间复杂度对比

11.5 实验

11.5.1 **样例分析**: *Miller − Rabin Primalty Test*

由于 M-R 素性测试是随机算法，过程中 a 是通过随机取数得到的。而根据 Wikipedia 的资料，在以下情况下算法可以被保证为确定性算法。

- if n < 1,373,653, it is enough to test a = 2 and 3.
- if n < 9,080,191, it is enough to test a = 31 and 73.
- if n < 4,759,123,141, it is enough to test a = 2, 7, and 61.
- if n < 2,152,302,898,747, it is enough to test a = 2, 3, 5, 7, and 11.

下面模拟 M-R 素性测试的过程 [表 11.1].
输入：$n = 1001$. 返回结果为 False.

a	x	s	d
-	-	3	125
2(Randomized)	1	-	-
81	8	-	-

表 11.1: M-R PT: n=1001

11.5.2 样例分析: 素数筛

输入：一个长度为 10 的序列 4,6,7,8,9,11,12,13,14,15.[表 11.2]

cs	sieve	all	primes
2,3,4,6,8,10,9,12,15	t,t,t,t,t,t,t,t,t	-	-
2,3,4,6,8,10,9,12,15	f,f,t,f,f,t,f,t,f,f	(2,t),(5,t),(7,t)	7,11,13

表 11.2: Prime Sieve

11.6 总结

素性测试实验，我使用 $Miller-Rabin\ Primalty\ Test$ 进行素性测试，使用线性筛法进行给定序列的素数筛选。前者的复杂度可以控制在 $O(k)$，后者的复杂度为 $W = O(nlgn), S = O(lgn)$。

第十二章　实验课总结与心得体会

至此，实验课九个实验已经全部列出报告，现总结九个实验的结论如下。

1. 快速排序是非确定性算法，受糟糕条件影响最小的是随机选取主元策略。前两种策略受坏情况影响较大，最坏 $W \in \Theta(n^2)$，正常 $W \in \Theta(nlgn)$，而随机选取策略较稳定，$W = O(nlgn),\ S = O(lg^2 n)$。

2. 求解单源最短路可采用 Dijkstra 算法，该算法只能串行执行，Bellman-Ford 算法可以弥补这个缺陷。前者 $W = S \in O(mlgn)$，后者 $W \in O(nmlgn),\ S \in O(nlgn)$。

3. 求闭合串中最大括号长度，可以使用暴力方法、非递归栈结构和递归分治策略三种方法，复杂度分别为 $W \in O(n^2), W \in O(n), S \in O(nlgn)$。

4. 天际线问题，可以用线段树法、扫描线法和分治法。本实验中讨论了扫描线法和分治法，前者复杂度是 $W = S \in O(nlgn)$，后者复杂度是 $W \in O(nlgn), S\ O(lg^2 n)$。

5. 判断一个字符串序列是否是括号匹配，可以使用只能串行执行的 Iterate 方法和可并行执行的 Scan 方法来实现括号匹配算法，前者 $W = S = O(n)$，后者 $W = O(n),\ S = O(lgn)$.

6. 计算两个大整数的和、差、积，可以使用传统串行的竖式加法、减法和乘法，还有一种基于分治策略的可并行乘法，加减法复杂度为 $O(n)$，串行乘法复杂度为 $O(n^2)$，并行复杂度为 $S \in O(lg^2 n)$。

7. 求出给定无向无权连通图中所有割点和割边的数目，可以 Tarjan 算法进行深度优先搜索实现，最优实现方式的复杂度为 $W = S \in O(m + n)$。

8. 区间最值查询问题，使用在线处理方法 Sparse Table 算法，预处理开销为 $O(nlgn)$，查询开销为 $O(1)$。

9. 素性测试实验，可以使用 $Miller - Rabin\ Primalty\ Test$ 进行素性测试，使用线性筛法进行给定序列的素数筛选。前者的复杂度可以控制在 $O(k)$，后者的复杂度为 $W = O(nlgn), S = O(lgn)$。

通过一学期的学习，我学习了并行编程思维、数据结构、分析算法复杂度的方法、算法设计的技巧和函数式编程语言特性，收获匪浅。在实验报告的最后要特别感谢陆枫老师的精彩授课和悉心指导，也感谢学院为 ACM 班从 CMU 引进这门令人大开眼界的课程。