Санкт-Петербургский государственный университет

Временная сложность оптимальной сортировки

Курсовая работа
студента II курса
направление «Математика»
01.03.01.
группы 15.Б02–мм
очной формы обучения
Золотова Бориса Алексеевича

Научный руководитель:
Доцент, кандидат физико-математических наук,
Булычев Дмитрий Юрьевич

Санкт-Петербург
2017 год

Введение

Постановка задачи

Пусть дано натуральное число n. Пусть также задан список из n различных элементов, на которых введено отношение порядка — и больше нам про них ничего не известно. Задача сортировки данного списка заключается в том, чтобы расположить эти элементы в порядке возрастания.

Через $S(n)$ будем обозначать максимальное количество сравнений, которое выполнит оптимальный алгоритм сортировки списка из n элементов. Это значит, что не существует алгоритмов, которые в худшем случае работают быстрее.

Задача поиска $S(n)$ рассмотрена в деталях в книге Дональда Э. Кнута [art-of-programming]. В частности, в ней приведены значения $S(n)$ для первых натуральных n.

В частности, в книге Кнута показано, что при $n < 12$ значение $S(n)$ в точности равно тривиальной нижней оценке, которую можно дать на это значение — $\lceil \log_2(n!) \rceil$ (мы докажем её в теореме 1). В случае $n = 12$ книга приводит идею доказательства того, что не существует алгоритмов сортировки, завершающихся за $\lceil \log_2(12!) \rceil = 29$ шагов. Эта идея опирается на довольно сложный и нетривиальный перебор.

Цель данной работы — сделать из доказательств перебором, аналогичных тем, к которым нас отсылает Кнут, формальные доказательства несуществования алгоритмов заданной сложности, и затем верифицировать их с помощью компьютера. В частности, верификация будет означать, что рассмотрены все необходимые случаи и это сделано без ошибок.

Инструментарий

Для формализации и верификации доказательств мы будем использовать язык Agda. Если посмотреть с одной стороны — это немного странный функциональный язык программирования. Но с другой — его система типов настолько мощна, что её можно использовать для доказательства математических утверждений. Делается это при помощи изоморфизма Карри-Ховарда — соответствия, связывающего между собой типы в языках программирования и логические формулы.

Например, рассмотрим функцию $\lambda x.x$, возвращающую свой аргумент неизменным. Её тип — `∀ a . a → a`. С другой стороны, мы можем трактовать этот тип как формулу исчисления предикатов (а именно — импликацию $a \Rightarrow a$), и функцию — как доказательство этой формулы. Подробнее об изоморфизме Карри—Ховарда и о связанных с ним понятиях можно узнать в книге М. Соренсена [curry-howard].

В дальнейшем — когда доказательства теорем приводятся в виде исходного кода на Agda — необходимо мыслить практически каждый вводимый нами тип, как некоторое утверждение, которому требуется доказательство.

С элементарными примерами доказательств, верифицируемых посредством Agda, читатель может ознакомиться в руководстве [proving-equality].

Также примером работы с верификацией в Agda может послужить статья на портале Habrahabr [verified-qsort]: в ней описывается алгоритм быстрой сортировки списка и доказывается, что результат его работы представляет из себя переставленный исходный список, в котором элементы упорядочены требуемым образом.

Основные результаты

1) На языке Agda было формализовано понятие алгоритма сортировки списка значений, а также был построен инструментарий, позволяющий доказывать или опровергать существование алгоритма, сортирующего данный список;

2) Была написана программа на языке Haskell, позволяющая по данному списку и данному s (наихудшей временной сложности сортировки списка) либо построить алгоритм сортировки, использующий не более s сравнений, либо привести доказательство, опровергающее существование такого алгоритма.

Однако, сам размер доказательства, получаемого нашим способом, достаточно велик: $\Omega\left((n^2)^s \cdot n!\right)$. Поэтому, например, уже для $n = 6$ верификация на Agda уже не завершается за обозримое время на среднем персональном компьютере.

Также было введено понятие аномального списка: списка, сложность сортировки которого превышает $\log_2(n!)$. Если рассматривать естественную задачу поиска $S(n)$, наименьший такой список получается при $n = 12$, но в силу крайне большой вычислительной сложности перебора всех сортирующих алгоритмов он не может быть разобран нами с помощью подручных средств.

Тем не менее, нам удалось его «сымитировать»: найден список из 30 перестановок такой, что все перестановки в нём нельзя распознать за 5 сравнений (хотя $30 < 2^5$). Доказательство того, что все его элементы действительно нельзя распознать за 5 сравнений, не опирается на принцип Дирихле и другие тривиальные соображения, а в полной мере демонстрирует принципы, заложенные нами в доказательство несуществования сортирующих алгоритмов заданной сложности.

Исходные коды всех программ, написанных в процессе создания данной работы, хранятся в репозитории на GitHub:

`https://github.com/boris-a-zolotov/`
`Computational-complexity-of-quickest-sort`

Содержание

Идея доказательства

Идея доказательства

Описывая перебор, который необходимо произвести, чтобы доказать несуществование алгоритма сортировки 12 элементов за 29 сравнений, доказательство в [art-of-programming] опирается на следующую идею. Пусть мы хотим доказать, что алгоритмов сортировки сложности d нет — доказательство будем проводить по индукции.

Базой такого доказательства будут списки из перестановок l, такие, что $2^d < \operatorname{length} l$, доказательство отсутствия алгоритма для которых следует из принципа Дирихле.

Переход же будет строиться на разборе всех возможных сравнений, которые мы можем проверить. Пусть мы хотим показать, что алгоритм сортировки сложности d не может быть построен. Для этого мы рассмотрим все возможные алгоритмы, и для корневого сравнения каждого из них найдём контрпример. Им будет список перестановок, который данным сравнением разбивается на такие подсписки, что хотя бы один из них не может быть отсортирован за $d - 1$ шаг.

Соответственно, при доказательстве перехода для сложности d нам нужно предъявить $n(n-1) \,/\, 2$ доказательство невозможности сортировки сложности $d - 1$.

Далее мы рассмотрим эту идею в подробностях: нам необходимо определить, как именно мы будем мыслить алгоритмы сортировки и перебирать сравнения.

Алгоритм сортировки

Пусть дан список $a = [a_0 \ldots a_{n-1}]$ из n элементов, попарно различных и попарно сравнимых. Его в таком случае можно отождествить со списком чисел от 0 до $n - 1$ — то есть, на самом деле, с перестановкой этих чисел.

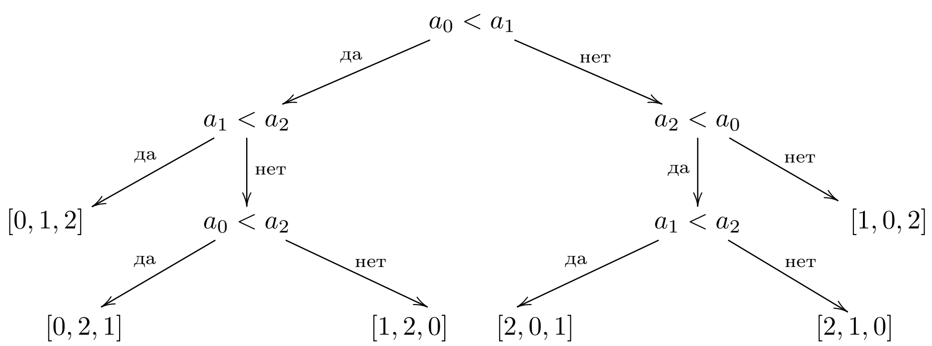
Наша задача — отсортировать список a, то есть, выяснить, с какой перестановкой его можно отождествить, задавая только вопросы вида «больше ли элемент a_i, чем элемент a_j?». Если мы поймём это, то будем знать, как переставить элементы списка a так, чтобы получить список $[0 \ldots (n - 1)]$.

Понятно, что в процессе сравнения элементов списка следующие вопросы зависят от ответов, полученных на предыдущие. Например, если мы выяснили, что $a_0 < a_1$ и $a_1 < a_2$, то вопрос «меньше ли a_0, чем a_2?» не имеет смысла.

Алгоритмом сортировки будем называть дерево, в узлах которого — вопросы вида «правда ли, что $a_i < a_j$?», $i, j \in \{0 \ldots (n - 1)\}$; при этом для каждого узла левое поддерево соответствует ответу «да» на этот вопрос, а правое — ответу «нет». Вопросы типа «$a_i < a_i$?» мы будем запрещать — они бессмысленны, так как одно из поддеревьев должно получаться пустым.

В листьях же алгоритма находятся перестановки — причём перестановка помещается в лист, только если вопросы в её предках и выбранные ответы на них однозначно её определяют (она единственная соответствует им всем).

Алгоритм 1. *Алгоритм сортировки для списка из трёх элементов (или, то же самое, для списка из 0, 1, 2).*



Характеристики алгоритма сортировки

Заметим, что поддеревья алгоритмов сортировки можно мыслить как частные алгоритмы сортировки, выбирающие ответ не из полного списка перестановок, а из ограниченного: из перестановок, для которых верно соответствующее сравнение.

Например, левое поддерево Алгоритма 1 сортирует список

$$\big[[0, 1, 2], [0, 2, 1], [1, 2, 0]\big].$$

Тогда становится понятно, что находится в листьях дерева: алгоритмы сортировки для списка из одной-единственной перестановки.

Если же алгоритм — узел, в котором задан вопрос «$a_i < a_j$?», и он должен выбрать ответ из некоторого списка перестановок (назовём его l), то его левый потомок сортирует список $[x \in l \mid x_i < x_j]$, а правый — соответственно, список $[x \in l \mid x_i \not< x_j]$.

В дальнейшем мы также будем говорить, что алгоритм *распознаёт* перестановки в том списке, который он сортирует.

Глубиной и количеством листьев алгоритма называются его глубина и количество листьев как дерева. Например, для алгоритма 1 глубина равна 3, количество листьев — 6.

Несложно видеть, что глубина алгоритма равна наибольшему количеству вопросов, которое он предписывает задать для выяснения перестановки. То есть, в точности его **сложности в наихудшем случае** — таким образом, нам удалось формализовать это понятие.

Количество листьев же примерно соответствует количеству элементов списка, перестановки из которого распознаёт алгоритм. В дальнейшем нами будет доказана теорема о том, что длина этого списка не превосходит количества листьев в алгоритме.

Оценка сложности алгоритма

Докажем тривиальную оценку на $S(n)$:

Теорема 1. $S(n) \geq \log_2(n!)$

Доказательство. Посмотрим на оптимальный алгоритм сортировки — он представляет из себя дерево, в листьях которого находятся перестановки. Глубина этого дерева — $S(n)$. Значит, в нём не более $2^{S(n)}$ листьев. Перестановок же в точности $n!$. Отсюда

$$2^{S(n)} \geq n!$$

$$S(n) \geq \log_2(n!).$$

□

В дальнейшем аналогичный факт будет доказан нами для вновь введённой формализации понятий алгоритма и его сложности. Кроме того, для всех последующих теорем в скобках рядом с их номерами мы будем писать их имена в соответствующих файлах Agda.

О несуществовании алгоритмов

Теперь мы пытаемся понять, существуют ли алгоритмы заданной глубины, распознающие все перестановки данного списка. По [art-of-programming], когда их может не быть?

1) Если заданная глубина равна d, а длина списка перестановок строго больше, чем 2^d — тогда существование алгоритма противоречило бы теореме 1.

2) Если при проверке всех возможных вопросов, которые могут быть заданы на данном шаге, выясняется, что хотя бы в одном поддереве для каждого из них мы рано или поздно придём к противоречию с теоремой 1. Тогда также можно заключить, что нужного алгоритма сортировки не найдётся.

В этом и будет заключаться наша техника доказательства несуществования алгоритмов.

Аномальный список

```
https://github.com/boris-a-zolotov
/Computational-complexity-of-quickest-sort
/blob/master/anomal_jumbo_list.agda
```

Нам будет интересно немного обобщить исходную задачу и рассмотреть алгоритмы сортировки произвольных списков перестановок — не только списков всех перестановок n элементов.

В частности, такое обобщение интересно тем, что для демонстрации принципов доказательств, которые мы строим, нам необходимы нетривиальные примеры сортируемых списков, на которых не работает принцип Дирихле и при этом верны строгие неравенства — например, $S(12) > \lceil \log_2(12!) \rceil$.

Тем не менее, из-за огромной алгоритмической сложности задачи, со списками длины 12! очень сложно работать. Аномальные списки дадут нам такие необходимые примеры меньшей длины.

Определение. *Список из перестановок l будем называть аномальным, если существует такое натуральное n, что длина l меньше 2^n, но тем не менее все перестановки из l нельзя распознать никаким алгоритмом сортировки, имеющим глубину не более n.*

Основную задачу можно переформулировать во введённых терминах так: проверить, является ли список всех перестановок n элементов аномальным.

Пример 1. *Список длины 30, который тем не менее невозможно распознать целиком за 5 сравнений (пусть даже $30 < 2^5$). Это его свойство снабжено формальным доказательством, с которым можно ознакомиться по ссылке.*

```
(0,1,2,3,4,5);(0,1,2,3,5,4);

(1,0,2,3,4,5);(1,0,3,2,4,5);(1,2,0,3,4,5);
(1,2,3,0,4,5);(1,3,0,2,4,5);

(1,0,2,3,5,4);(1,0,3,2,5,4);(1,2,0,3,5,4);
(1,2,3,0,5,4);(1,3,0,2,5,4);

(2,0,1,3,4,5);(2,0,3,1,4,5);(2,3,0,1,4,5);

(2,0,1,3,5,4);(2,0,3,1,5,4);(2,3,0,1,5,4);

(3,0,1,2,4,5);(3,0,2,1,4,5);(3,1,0,2,4,5);
(3,1,2,0,4,5);(3,2,0,1,4,5);(3,2,1,0,4,5);

(3,0,1,2,5,4);(3,0,2,1,5,4);(3,1,0,2,5,4);
(3,1,2,0,5,4);(3,2,0,1,5,4);(3,2,1,0,5,4).
```

Теперь наша задача — зафиксировать и формализовать всё то, что мы только что описали.

Равенство и неравенство натуральных чисел

```
github.com/boris-a-zolotov
/Computational-complexity-of-quickest-sort
/blob/master/equation_properties.agda
```

Если читатель не знаком близко с техникой доказательств в таких языках, как Agda, — в этом разделе мы приводим примеры формальных доказательств простых свойств натуральных чисел. В частности, мы докажем, что равенство является отношением эквивалентности и $\leq$ является отношением порядка. Также будут доказаны некоторые другие естественные свойства натуральных чисел, которые в Agda неочевидны и нуждаются в формальном доказательстве.

За равенство элементов в Agda отвечает тип «$a \equiv b$», соответствующий посредством изоморфизма Карри—Ховарда утверждению «выражения для a и для b одновременно β-редуцируются к одной и той же нормальной форме».

Тип $a \equiv b$ в зависимости от a и b может быть либо пуст, либо населён единственным элементом `refl`. Если Agda может, просто редуцировав выражения для a и для b, получить их общую нормальную форму (как, например, в случае $a = 2 + 1$, $b = 3$), то утверждение $a \equiv b$ можно посчитать «очевидным», и успешно верифицируется доказательство:

```
proof : 2 + 1 ≡ 3
proof = refl
```

В более сложных случаях (когда, например, выражения для a и для b зависят от других параметров) требуется, соответственно, более развёрнутое доказательство. Но при каждом конкретном значении параметров любое такое сложное доказательство окажется эквивалентно `refl`-у: ведь если два выражения равны, то их можно β-редуцировать к одной и той же форме.

Теорема 2. (eq_comm) *Отношение равенства симметрично: для всех a, b выполнено $a \equiv b \Longrightarrow b \equiv a$.*

Доказательство. Нам нужно по любому доказательству равенства $a \equiv b$ научиться строить доказательство равенства $b \equiv a$. Доказательством равенства может быть только `refl`, поэтому единственный вариант для нас — по `refl`-у возвращать его же. Получается:

```
eq_comm : ∀ {P : Set} {m n : P} → (m ≡ n) → (n ≡ m)
eq_comm refl = refl
```

Это и является утверждением и доказательством данной теоремы. □

Теорема 3. (eq_trans) *Отношение равенства транзитивно: для всех a, b, c выполнено $a \equiv b$, $b \equiv c \Longrightarrow a \equiv c$.*

Данная теорема доказывается аналогично: единственно возможные доказательства, которые могут поступить нам на вход — это `refl`-ы, вернуть в таком случае мы тоже должны будем просто `refl`.

Абсолютно аналогично доказываются две следующие теоремы:

Теорема 4. (eq_cong) $\forall\ P, Q\ \ \forall\ a, b \in P\ \ \forall\ f\colon P \to Q$

$$a \equiv b \implies f(a) \equiv f(b).$$

Последнее свойство также называется конгруэнтностью.

Теорема 5. (preserv-+) $\forall\ m, n, x, y \in \mathbb{N}$

$$m \equiv n,\ x \equiv y \implies m + x \equiv n + y.$$

Для нестрогого сравнения ($\leq$) нам нужно доказать то, что оно является отношением порядка: его рефлексивность, антисимметричность и транзитивность. При этом доказательство утверждения «$a \leq b$» для двух чисел a, b может иметь один из двух видов:

- `z≤n`, если $a = 0$, b — произвольное;
- `s≤s p`, если $a, b > 0$, и `p` — доказательство того, что $a - 1 \leq b - 1$.

Теорема 6. (trans_≤) *Для любых натуральных чисел a, b, c выполнено $a \leq b,\ b \leq c \Longrightarrow a \leq c$.*

Доказательство. Как обычно, по паре доказательств для $a \leq b$ и $b \leq c$ нам нужно построить доказательство для $a \leq c$. Заметим, что если первое доказательство имеет вид `z≤n`, то непременно $a = 0$ (и поэтому не превосходит любое c), поэтому мы можем вернуть `z≤n`. Это будет базой индукции.

Переход: если a, b, c строго больше нуля, то первое доказательство имеет вид `s≤s p`, а второе `s≤s q`. Тогда рассмотрим числа $a-1$, $b-1$, $c-1$ и пару доказательств `p`, `q`. По индукционному предположению, мы можем построить доказательство `P` того, что $a-1 \leq c-1$. Тогда `s≤s P` и будет доказательством факта $a \leq c$.

Получается:

```
trans_≤ : ∀ {a b c : ℕ} → (a ≤ b) → (b ≤ c) → (a ≤ c)
trans_≤ z≤n _ = z≤n
trans_≤ (s≤s p) (s≤s q) = s≤s (trans_≤ p q)
```

□

Теорема 7. (antisymm) *Отношение $\leq$ антисимметрично: для любых двух натуральных чисел a, b выполнено:*

$$a \leq b\ \wedge\ b \leq a \implies a \equiv b$$

Доказательство. Рассмотрим несколько случаев:

1) Если $a = b = 0$, то вне зависимости от конкретных доказательств того, что они не больше друг друга, доказательством утверждения $0 \equiv 0$ будет очевидно являться `refl`. Его и вернём — это будет базой индукции.

2) Если ровно одно из двух чисел a, b равно нулю, а другое — нет, то не сможет быть доказуемым хотя бы одно из пары утверждений $a \leq b$, $b \leq a$. Поэтому такой случай, очевидно, невозможен — в Agda в таком случае необходимо написать круглые скобки; это будет верифицировано.

3) Индукционный переход: если $a > 0$, $b > 0$, $a \leq b$ и $b \leq a$, то рассмотрим числа $a - 1$ и $b - 1$. Для них выполнено то же самое: $a - 1 \leq b - 1$, $b - 1 \leq a - 1$. Но тогда мы можем доказать $a - 1 \equiv b - 1$. В свою очередь, по теореме 4 (взяв $f(x) = x + 1$) имеем $(a - 1) + 1 \equiv (b - 1) + 1$ — то есть, $a \equiv b$.

Получаем:

```
antisymm : (m n : ℕ) → m ≤ n →
    n ≤ m → n ≡ m
antisymm 0 0 p q = refl
antisymm (suc x) 0 ()
antisymm 0 (suc x) p ()
antisymm (suc x) (suc y) (s≤s p) (s≤s q) =
    eq_cong (suc) (antisymm x y p q)
```

□

Аналогично доказывается

Теорема 8. (noteq) *Для любых $m, n \in \mathbb{N}$, если $m < n$, то не верно, что $m = n$.*

Доказательство.

```
noteq : (m n : ℕ) → m < n → m ≡ n → ⊥
noteq 0 0 ()
noteq (suc x) 0 ()
noteq 0 (suc x) p ()
noteq (suc x) (suc y) (s≤s p) q =
    noteq x y p (eq_cong (pred) q)
```

□

Перестановки

```
https://github.com/boris-a-zolotov
/Computational-complexity-of-quickest-sort
/blob/master/permutation_definition.agda
```

В дальнейшем нам пригодится понятие перестановки: перестановка длины n — это список из n элементов, в котором все элементы — различные целые числа от 0 до $n - 1$. Мы для простоты не будем включать в определение для Agda требование того, что все элементы в списке обязаны быть различными, а добавим его в явном виде позже, когда это свойство нам понадобится.

Для начала определим тип «список длины n»:

```
data Nlist (A : Set) : ℕ → Set where
    «» : Nlist A 0
    _-_ : {n : ℕ} → A → Nlist A n → Nlist A (suc n)
```

После этого скажем, что нужный нам тип «перестановка» — список длины n, составленный из целых чисел от 0 до $n-1$:

```
BorderedNlist : ℕ → Set
BorderedNlist n = Nlist (Fin n) n
```

Тип `Fin` n (целые числа, строго меньшие n) строится в соответствии с аксиоматикой Пеано: элемент `zerof` — ноль — принадлежит типу `Fin` n при всяком $n \geq 1$, а конструктор `sucf` — «следующее число» — делает из элемента типа `Fin` n элемент типа `Fin` $n+1$. Таким образом, единица в типе `Fin` 3 будет выглядеть как `sucf zerof`, где `zerof` принадлежит типу `Fin` 2.

Тогда, например, перестановка $[0, 2, 1]$ будет теперь выглядеть так:

```
p021 : BorderedNlist 3
p021 = zerof - ((sucf (sucf zerof)) - ((sucf zerof) - «»))
```

Для типа «перестановка» можно определить функцию `nth`, возвращающую по номеру элемент в перестановке под этим номером. Также нам пригодится предикат `listCmp i j p`, отвечающий на вопрос: «верно ли, что i–ый элемент перестановки `p` меньше, чем её j–ый элемент?». С тем, как описаны эти функции, читатель может ознакомиться в исходном коде.

Имея предикат, можно определить соответствующую функцию высшего порядка:

```
cmpFilter : {n : ℕ} → (i j : Fin n) →
    (l : List (BorderedNlist n)) → List (BorderedNlist n)
cmpFilter {n} i j l = filter (listCmp {n} i j) l
```

Тип Unicalized

```
https://github.com/boris-a-zolotov
/Computational-complexity-of-quickest-sort
/blob/master/final_length.agda
```

Пусть алгоритм сортировки расставляет в свои листья перестановки (длины n) из списка l, на котором он определён. Для доказательства некоторых важных свойств алгоритмов нам потребуется явное доказательство того, что все элементы l являются перестановками в смысле математического определения (как и обещано в предыдущем параграфе) — то есть

$$\forall\ k \in \{1 \ldots (\text{length}\ l) - 1\}\ \ \forall\ i, j \in \{1 \ldots (n-1)\}$$
$$i \neq j \implies (l_k)_i \neq (l_k)_j.$$

Этому утверждению (опять же, посредством изоморфизма Карри—Ховарда) будет соответствовать тип `Unicalized`, определённый следующим образом:

```
data Unicalized : {n : ℕ} →
    List (BorderedNlist n) → Set
where
    u-nil : {n : ℕ} → Unicalized {n} []
    _u-cons_ : {n : ℕ} →
        {l : List (BorderedNlist n)} → {x : BorderedNlist n} →
        ((i j : Fin n) → (i ≢ j) → nth x i ≢ nth x j) →
        Unicalized l → Unicalized (x :: l)
```

Элемент типа `Unicalized l` на самом деле является списком, элементы которого соответствуют элементам `l`. Каждый элемент этого списка есть доказательство того, что соответствующий ему элемент `l` — перестановка.

Несложно доказать следующую теорему:

Теорема 9. (projUnic) *Если список l_1 является подпоследовательностью (подсписком) списка l_0, то* `Unicalized` l_0 *влечёт* `Unicalized` l_1.

Очевидно, что если к любому списку применить функцию `filter` по любому предикату, то мы получим его подсписок. На этом основании из теоремы 9 очевидно следует

Теорема 10. (filterUnic) *Если для списка l, состоящего из перестановок длины n, верно* `Unicalized` l, *то*

$$\forall\ i, j \in \{1 \ldots (n-1)\} \ \ \textit{имеет место} \ \ \texttt{Unicalized}\ (\texttt{cmpFilter}\ i\ j\ l).$$

Приведём пример утверждения, для доказательства которого необходимо наличие `Unicalized` для исходного списка — и которое, легко понять, не верно в противном случае:

Теорема 11. (FILTERLENGTH) *Если для списка l доказано* `Unicalized` l, *то для любых $i, j \in \{1 \ldots (n-1)\}$, $i \neq j$ выполнено*

$$\text{length}\ l \quad \leq \quad \text{length}\ (\texttt{cmpFilter}\ i\ j\ l) + \text{length}\ (\texttt{cmpFilter}\ j\ i\ l).$$

С доказательствами этих и других утверждений читатель может ознакомиться в исходном коде.

Алгоритм сортировки и его характеристики

Алгоритм

```
https://github.com/boris-a-zolotov
/Computational-complexity-of-quickest-sort
/blob/master/algorithm.agda
```

В первом разделе мы определили алгоритм сортировки. На Agda это определение будет выглядеть следующим образом:

```
data Alg : {n : ℕ} → List (BorderedNlist n) → Set where
    leaf : {n : ℕ} → (b : BorderedNlist n) → Alg (b :: [])
    node : {n : ℕ} → {l : List (BorderedNlist n)} →
        (i j : Fin n) → (i ≢ j) →
        Alg (cmpFilter i j l) → Alg (cmpFilter j i l) → Alg l
```

Действительно, всякий алгоритм сортировки — это либо «лист» (когда мы наверняка нашли единственную перестановку), либо «узел»: когда мы сравниваем элементы на двух фиксированных позициях в каждой из перестановок, и для каждого из подсписков (где $(l_k)_i < (l_k)_j$, и где $(l_k)_i \not< (l_k)_j$) у нас есть алгоритм сортировки.

С примерами описания алгоритмов сортировки читатель может ознакомиться в исходном коде.

Глубина и количество листьев

```
https://github.com/boris-a-zolotov
/Computational-complexity-of-quickest-sort
/blob/master/alg2n.agda
```

Глубина алгоритма сортировки (она также была рассмотрена нами в первом разделе) определяется в Agda следующим образом:

```
depth : {n : ℕ} →
    {l : List (BorderedNlist n)} → Alg l → ℕ
depth (leaf b) = 0
depth (node i j p a1 a2) = suc ((depth a1) ⊔ (depth a2))
```

Количество листьев для алгоритма вводится аналогично:

```
leafnum : {n : ℕ} →
    {l : List (BorderedNlist n)} → Alg l → ℕ
leafnum (leaf b) = 1
leafnum (node i j p a1 a2) = (leafnum a1) + (leafnum a2)
```

Как уже было сказано, количество листьев алгоритма определяет, сколько перестановок он может обработать. Это и утверждает следующий факт, для доказательства которого потребовалась теорема 11:

Теорема 12. (lenLeafNum) *Для любого алгоритма A, принадлежащего типу* `Alg` *l, при условии* `Unicalized` *l выполнено*

$$\text{length } l \ \leq \ \texttt{leafnum } A.$$

Связь между ними

С использованием теоремы 12 доказан крайне полезный в дальнейшем аналог теоремы 1:

Теорема 13. (len2depth)

$$\forall\ l \quad \forall\ A \in \texttt{Alg}\ l$$

$$\texttt{Unicalized}\ l \implies \text{length}\ l \ \leq\ 2^{\text{depth}\,A}.$$

Доказательства всех упомянутых теорем приведены в исходном коде.

Тип noexist и отсутствие алгоритмов

```
https://github.com/boris-a-zolotov
/Computational-complexity-of-quickest-sort
/blob/master/noexist.agda
```

Пусть дан список l, состоящий из каких-то перестановок n элементов. Пусть также имеется доказательство того, что его элементы — именно перестановки, то есть составлены из неповторяющихся чисел $0 \ldots n-1$: `Unicalized` l. Пусть также дано число d.

Если мы хотим доказать, что не существует алгоритмов глубины не более чем d, распознающих все перестановки в списке l, то для этого мы должны построить функцию, которая по каждому алгоритму глубины d строит противоречие. Для этого мы воспользуемся значением типа `noexist`, которое будет содержать результаты перебора и рассмотрения всех возможных алгоритмов.

Значение типа `noexist` может быть одного из двух видов. С одной стороны, алгоритма может не существовать попросту из-за принципа Дирихле (не выполнено условие теоремы 13): этой ситуации будет соответствовать значение `basa`, несущее с собой рассматриваемый список, глубину и доказательство того, что $2^{\text{depth}} < \text{length}\, l$.

С другой стороны, мы можем проверить все возможные сравнения, которые можно только рассмотреть, и для каждого из них получить, что для левого или для правого из двух подсписков нет алгоритма глубины $d-1$. Этой ситуации будет соответствовать значение `pereh`, опять же несущее с собой список, глубину и функцию, возвращающую по каждому из возможных сравнений `noexist` для левого <u>или</u> для правого из подсписков-сыновей.

Логической связке «или» соответствует дизъюнктное объединение типов. Если мы хотим доказать утверждение $A \vee B$, мы должны указать, какое конкретно из них доказываем — и построить его доказательство. Дизъюнктному объединению в Agda приписан символ $\uplus$.

Затем из наличия значения типа `noexist` нам надо будет доказать уже сам факт отсутствия алгоритма. В частности, наличие `noexist`-а для списка из всех $n!$ перестановок длины n, должно повлечь то, что произвольный список из n элементов нельзя отсортировать за d шагов.

Таким образом, тип `noexist` может быть определён нами следующим образом:

```
data noexist : {n : ℕ} → (l : List (BorderedNlist n)) →
    (depth : ℕ) → Set
where
  basa : {n : ℕ} → (l : List (BorderedNlist n)) → (d : ℕ) →
    (2 ^ d ℕ< length l) → noexist l d
  pereh : {n : ℕ} → (l : List (BorderedNlist n)) → (d : ℕ) →
    ((i j : Fin n) → (i ≢ j) →
      (noexist (cmpFilter i j l) d ⊎ noexist (cmpFilter j i l) d))
    → noexist l (suc d)
```

Может показаться странным, что при $n = 1$ определение `noexist` позволяет построить значение `pereh`, не имея при этом никаких конкретных доказательств. Однако задача сортировки списка из одного элемента не имеет содержательного смысла. Поэтому в дальнейшем мы будем попросту запрещать случай $n = 1$.

Оказывается, что при введённом нами определении `noexist` для него можно сразу доказать одно интересное свойство.

Теорема 14. (noexistBigLength) *Если дан список l, состоящий из каких-то перестановок длины n, при $n \geq 2$, и для какого-то d нашёлся элемент, населяющий тип* `noexist` *l d, то длина l строго больше 1.*

Доказательство. Если элемент из типа `noexist` имеет вид `basa` l d p, тогда p — непременно доказательство того, что $2^d < \text{length } l$, где d — натуральное число. Тогда $1 \leq 2^d < \text{length } l$ — что и требовалось.

В противном случае, когда элемент из типа `noexist` выглядит как `pereh` l d f, посмотрим на $f(0, 1)$ — она укажет нам на какой-то `noexist` для списка, к которому был применён `filter`. Длина списка, очевидно, не увеличивается после фильтрации.

Так, спускаясь всё глубже в данный нам `noexist`, мы рано или поздно дойдём до `basa` — а в таком случае доказано, что длина много раз отфильтрованного списка всё равно строго больше единицы.

С формальным изложением доказательства читатель может ознакомиться в исходном коде. □

Как мы уже говорили, доказанный факт `noexist l d` не утверждает напрямую отсутствия алогритма данной глубины для данного списка. Однако несуществование алгоритма может быть выведено из существования `noexist l d`. Справедлива следующая теорема:

Теорема 15. (noexistNoAlg) *Пусть $n \geq 2$, и дан список l, состоящий из перестановок n элементов. Пусть также имеется натуральное число d. Тогда, при условии* `Unicalized` *l, выполнено*

$$\texttt{noexist}\ l\ d \implies (\exists\ a \in \texttt{Alg}\ l,\ \texttt{depth}\ a \leq d \ \Rightarrow\ \bot).$$

То есть, при условии `noexist`*-а из существования нужного алгоритма можно вывести ложь — что как раз и означает отсутствие алгоритмов.*

Доказательство. Пусть у нас есть алгоритм $a \in$ `Alg` l глубины не более d и элемент типа `noexist` l d. По определению типа `noexist` последний может принимать один из двух видов:

1) `basa` l d p, где p — доказательство того, что $2^d <$ length l. Вспомним, что у нас, кроме того, есть построенное по теореме 13 доказательство того, что length $l \leq 2^{\texttt{depth}\ a}$.

   Из неравенства `depth` $a \leq d$, доказательство которого дано нам по условию, можно вывести $2^{\texttt{depth}\ a} \leq 2^d$. Отсюда по теореме 6 (транзитивность неравенства) получаем length $l \leq 2^d$.

   Из получившихся двух утверждений ($x < y$ и $y \leq x$) можно вывести противоречие — $\perp$, как раз то, что нужно.

   В исходном коде это выглядит так:

```
noexistNoAlg : {n : ℕ} → {l : List (BorderedNlist n)} → {dep : ℕ}
  → (2 ℕ≤ n) → Unicalized l
  → noexist l dep → (a : Alg l) → (depth a ℕ≤ dep)
  → ⊥

noexistNoAlg neq u (basa l d p) a q = noComp
  {2 ^ d} {length l}
  p (trans_≤ (len2depth a u) (degs_ineq q))
```

2) `pereh` l d f, где функция f по данным ей позициям для сравнения показывает, для какого из поддеревьев при таком сравнении не найдётся алгоритма сортировки нужной глубины.

   Тогда рассмотрим ещё два случая: алгоритм, данный нам по условию, может быть либо листом, либо узлом. Если он лист — то немедленно получаем противоречие с теоремой 14, ведь длина соответствующего списка — 1. Соответствующий код на Agda:

```
noexistNoAlg neq u (pereh (.b :: []) d f) (leaf b) q = noteq
  1 (length (b :: []))
  (noexistBigLength neq (pereh (b :: []) d f)) refl
```

   Если же данный нам алгоритм — узел (`node` i j P a_1 a_2, где P — доказательство того, что $i \neq j$), то само значение типа `noexist`, которое нам дано, позволяет понять, к какому из поддеревьев этого алгоритма нужно перейти, чтобы получить противоречие.

   А именно — $f(i, j)$ указывает нам на `noexist` для одного из поддеревьев. Тогда противоречие получится из применения нашей теоремы к этому `noexist`-у и соответствующему алгоритму (a_1 или a_2).

   Также нам потребуются следующие вспомогательные факты: (1) глубина одновременно `noexist`-а и алгоритма для поддерева стала на единицу меньше; (2) утверждение `Unicalized`, в силу теоремы 10, сохранилось для отфильтрованного списка, на котором определён a_1 (или a_2).

   В исходном коде это выглядит так:

```
noexistNoAlg
  {n} {l} {suc d}
  neq u (pereh .l .d f) (node i j prf a1 a2) q
with (f i j prf)

... | inj₁ NOEX = noexistNoAlg
    neq (filterUnic i j l u) NOEX a1
    (decrease≤ {depth a1} {d}
      (trans_≤ (leftSon {n} {l} {i} {j} {prf} a1 a2) q))

... | inj₂ NOEXREV = noexistNoAlg
    neq
    (filterUnic j i l u) NOEXREV a2
    (decrease≤ {depth a2} {d}
      (trans_≤ (rightSon {n} {l} {i} {j} {prf} a1 a2) q))
```

□

Автоматическое построение доказательства

```
https://github.com/boris-a-zolotov
/Computational-complexity-of-quickest-sort/blob
/master/z_autoproof_unicalized.hs
```

```
https://github.com/boris-a-zolotov
/Computational-complexity-of-quickest-sort/blob
/master/z_autoproof_noexistOrAlg.hs
```

Пользователь ввёл список l из перестановок длины n (именно перестановок, то есть списков, элементы которых попарно различны) и глубину d. Мы хотим научиться автоматически решать задачу, которая была поставлена ранее: вывести либо алгоритм, распознающий все перестановки в списке за не более чем d сравнений, либо доказательство того, что таких алгоритмов нет.

Язык, с которым мы теперь работаем — Haskell. Мы будем активно пользоваться введёнными нами конструкциями `Unicalized` и `noexist`, которые очень удобно генерировать программно и которые помогают получить нужный результат.

Как вывести доказательство `Unicalized` для данного списка? Для этого нужно вывести доказательство попарной различности элементов каждой из составляющих его перестановок. Оно строится как функция, берущая на вход номера элементов в перестановке. Зафиксировав, что она не определена на равных аргументах и на аргументах не меньше n, несложно вывести нужные доказательства в остальных случаях. С конкретной реализацией читатель может ознакомиться в исходном коде.

Для поиска алгоритма введём следующий тип данных:

```
data AlgSearch =
        Leaf [Int] |
        Node Int Int AlgSearch AlgSearch |
        Noexist String | Wtf
```

Он хранит в себе меньше информации, чем конструкции, используемые нами в Agda — но для вывода кода, который потом будет верифицирован при помощи Agda, этой информации вполне хватит.

Алгоритм поиска построим следующим образом: пусть ему даны список l из перестановок и глубина d — тогда

1) Если $2^d < \text{length } l$, то алгоритма сортировки существовать не должно — возвращаем `Noexist` вместе со строкой, содержащей название соответствующего `noexist`-а (он будет построен с помощью конструктора `basa`), список, глубину и доказательство неравенства $2^d < \text{length } l$.

2) В противном случае, и если длина $l = [x]$ равна одному, алгоритм сортировки, несомненно, есть — возвращаем `Leaf` x.

3) Если длина l больше одного, то рассмотрим все возможные сравнения, которые можно произвести. Если хотя бы для одного из них получилось построить алгоритмы глубиной $d-1$, распознающие и перестановки, удовлетворяющие прямому сравнению, и перестановки, удовлетворяющие обратному — то мы делаем из этих двух алгоритмов узел и возвращаем его «наверх».

   Если же для каждого сравнения хотя бы для одного из фильтрованных списков и глубины $d-1$ алгоритм возвращает `Noexist`, то с использованием строк, которые вернулись вместе с ними, можно построить `noexist` для исходного списка и глубины d. Наверх вернётся `Noexist` вместе со строчкой–описанием `noexist`-а.

4) Заметим, что длина l вообще не может в интересующих нас случаях обратиться в ноль: если алгоритм поиска встречает список длины 1, он тут же возвращает `Leaf`. Если же пустой список получился в качестве одного из побочных при фильтре по сравнению, то другой побочный список имеет ту же длину, что и исходный, но глубину на единицу меньше — для него наверняка докажется `noexist`, если для исходного списка не нашлось алгоритма сортировки.

После завершения работы алгоритма поиска мы получим либо описание алгоритма, распознающего все перестановки — и его можно просто записать в файл для Agda, либо описание `noexist`-а — и чтобы из него получить доказательство отсутствия алгоритмов, нужно приписать в конец применение теоремы 15 к этому `noexist`-у.

Более подробно изучить реализацию можно, ознакомившись с исходным кодом.

Результаты

В результате была решена поставленная задача: по списку из перестановок и глубине мы получаем либо алгоритм глубины не больше заданной, распознающий все перестановки из списка, либо доказательство отсутствия таковых.

Например, возможности среднего персонального компьютера позволили нам с использованием нашего алгоритма в реальном времени (не более чем за 5–10 минут) доказать, что $S(5) = 7$.

Отсутствие алгоритмов глубины строго меньше, чем 7, было получено из неравенства $2^6 < 120$. Алгоритм глубины 7 был построен с помощью нашего алгоритма перебора, с ним можно ознакомиться по ссылке:

```
https://github.com/boris-a-zolotov
/Computational-complexity-of-quickest-sort
/blob/master/allperms_sorted.agda
```

Было успешно верифицировано с помощью Agda, что этот алгоритм распознаёт все перестановки списка $[0, 1, 2, 3, 4]$.

Поиск следующих значений $S(n)$ требует значительно бо́льших затрат времени и оперативной памяти. Проблема заключается в том, что построенный нами алгоритм перечисляет все перестановки длины n, что порождает большой объём данных, которые необходимо хранить в памяти.

Если оценить, сколько раз при построении получаемого перебором значения типа `noexist` выписываются все перестановки из исходного списка, несложно получить нижнюю оценку на асимптотику размера доказательства, конструируемого для списка l и глубины d:

$$\Omega\left((n^2)^s \cdot n!\right).$$

Возможно, помог бы другой способ хранения перестановок, который требует меньше памяти, но для которого по-прежнему несложно доказывать нужные факты про фильтры и сортировки. Например, способ хранения, связанный с числами в факториальной системе счисления, очень ёмкий — но для него сложно доказать то, за что в нашем случае отвечал просто определяемый тип Unicalized.

Также оптимизации нашего алгоритма могло бы помочь использование техники, введённой Д. Э. Кнутом в [art-of-programming], связанной с понятием *эффективности* алгоритма и пересчётом матриц для её вычисления.

Составление доказательств для этой работы включало в себя и неудачный опыт: например, попытки хранить перестановки в виде чисел в факториальной системе счисления. Также потерпело неудачу определение алгоритма сортировки «снизу», а не «сверху» — когда в узле хранится конкатенация списков потомков, а не списки для потомков определяются как применения функции `filter` к родительскому.

Таким образом, итогом данной работы явились построенная техника проверки существования алгоритмов сортировки заданной сложности (и эти результаты могут быть формально проверены) и примеры списков, для которых не существует алгоритмов сортировки конкретной глубины — из соображений менее тривиальных, чем принцип Дирихле.

Литература

[art-of-programming]
Кнут, Д. Э., «Искусство программирования для ЭВМ»: Монография [Текст]: в 4-х т. / Д. Кнут; Пер. с англ. Н.И. Вьюковой, В.А. Галатенко и др.; Под ред. Ю.М. Баяковского и В.С. Штаркмана. — Москва : Мир, 1978. — 844 с. — 3 т.: «Сортировка и поиск».

[curry-howard]
Sorensen, M. H. B. «Lections on the Curry—Howard Isomorphism» [Электронный ресурс] — M. H. B. Sorensen, P. Urzyczyn — Электронные данные. — Копенгаген: University of Copenhagen. — 1 документ в формате PDF.

[proving-equality]
ELTE Faculty of Informatics, «Proving Equality» [Электронный ресурс] — Электронн. текстовые дан. — Будапешт: ELTE. — Режим доступа: свободный, `http://people.inf.elte.hu/divip/AgdaTutorial/Functions.Equality_Proofs.html`;

[verified-qsort]
Д. Менделеев, «Верифицированный QuickSort на Agda» [Электронный ресурс] — Д. Менделеев. — Электронные текстовые данные. — 2012. — Режим доступа: свободный, `https://habrahabr.ru/post/148769/`.