Адрианов Н.М.
Иванов А.Б.

Алгоритмы и структуры данных

Timsort

Timsort

- Tim Peters (2002)
  `http://bugs.python.org/file4451/timsort.txt`
- Python 2.3, Java SE 7, Android 1.5
- В реальных ситуациях массивы данных часто содержат в себе упорядоченные подмассивы.
- Использует (модифицированую) процедуру слияния, суть алгоритма - в каком порядке выполнять слияние. Хочется сливать массивы близкого размера.
- Отсортированные куски храним в стеке, поддерживая некоторые условия (инвариант).
- Гибридный алгоритм: Небольшие подмассивы сортируются вставками.

Timsort: корректность алгоритма

Февраль 2015: обнаружена ошибка (инвариант алгоритма не соблюдался, что приводило к переполнению стека)

S. de Gouw, J. Rot, F. de Boer, R. Bubel, R. Hähnle:
Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

Предложено два варианта исправления
`https://bugs.openjdk.java.net/browse/JDK-8072909`

1) Исправить логику так, чтобы инвариант соблюдался
2) Увеличить размер стека

Python: исправлено по варианту 1
Java: исправлено по варианту 2

Timsort: корректность и сложность

Май 2018: Вариант исправления, выбранный в Java - некорректный (в работе 2015 года была ошибка).

N. Auger, C. Nicaud, C. Pivoteau, On the Worst-Case Complexity of TimSort. `https://arxiv.org/pdf/1805.08612.pdf`

`https://bugs.openjdk.java.net/browse/JDK-8203864`

В этой же работе доказано, что сложность алгоритма $O(n \log n)$ и даже $O(n + n \log \rho)$: Timsort – адаптивный алгоритм! Также ведет себя Natural MergeSort (Кнут, т.3).

`http://www-igm.univ-mlv.fr/~juge/slides/poster/ligm-2019.pdf`

Timsort: «run»

$$a_1 \le a_2 \le a_3 \le \ldots \le a_r$$

$$a_1 > a_2 > a_3 > \ldots > a_r$$

Выбираем число $\texttt{minrun} \in [32, 64)$ так, чтобы $N/\texttt{minrun}$ оказалось степенью двойки или немного меньше.

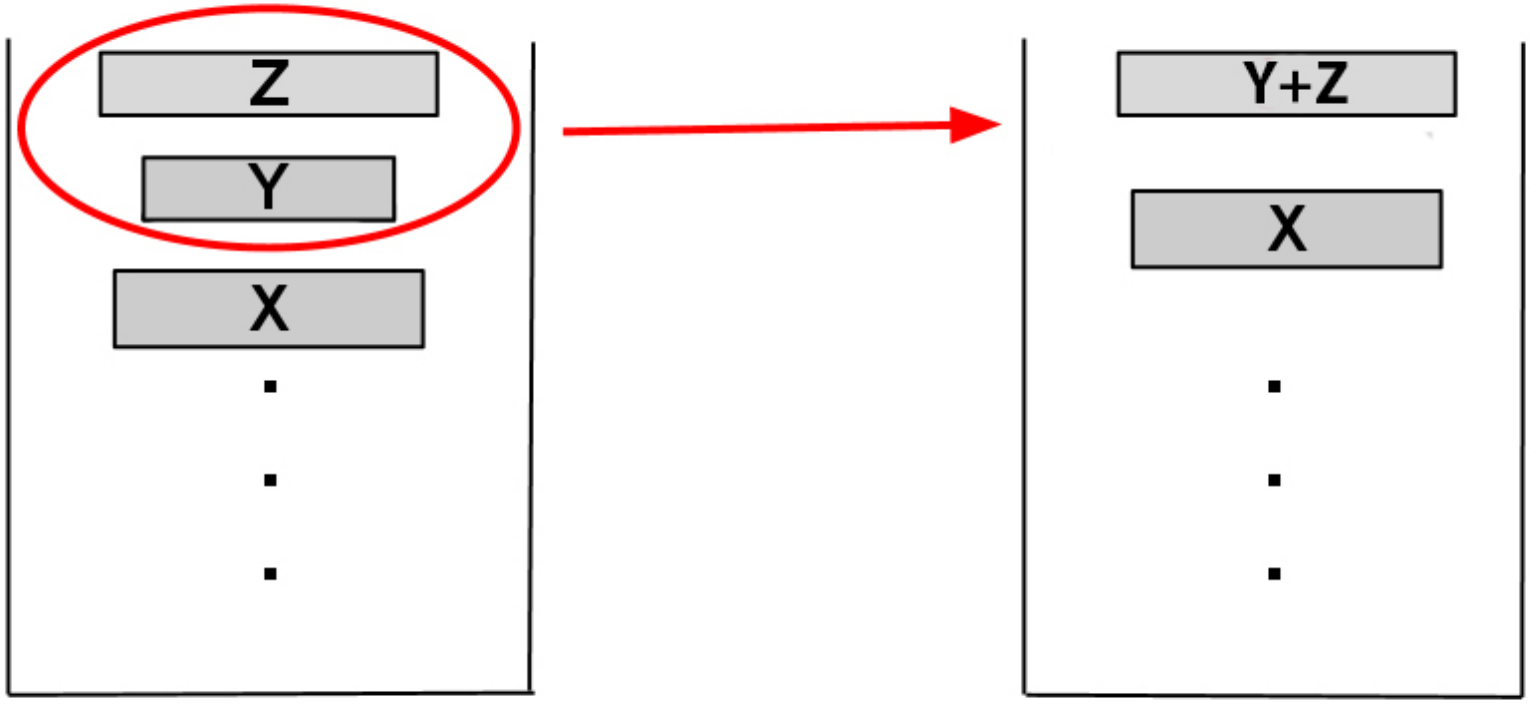
Пример: $N = 2112$
$\texttt{minrun} = 32$ – плохой $(2112 = 66 \times 32)$,
$\texttt{minrun} = 33$ – хороший $(2112 = 64 \times 33)$.

```
int r = 0;     // Becomes 1 if any 1 bits are shifted off
while (n >= minrun) { r |= (n & 1); n >>= 1; }
return n + r;
```

Timsort: общая схема

1. Определяем `minrun`, текущая позиция = 1
2. Находим run, начинающийся с текущей позиции
3. Если run убывающий – переворачиваем
4. Если run короче `minrun` – дополняем и сортируем вставками
5. Добавляем в стек (начало подмассива, длина)
6. Проверяем условия инварианта, если нарушены – выполняем слияние (см. след. слайд)
7. Если не дошли до конца массива – переходим к 2
8. Сливаем подмассивы, хранящиеся в стеке

Timsort: стек



- $X > Y + Z$
- $Y > Z$

Если $X \leq Y + Z$ – сливаем Y с минимальным из X и Z.
Если $Y \leq Z$ – сливаем Y с Z.

Timsort: еще раз про размер minrun

N = 320, minrun = 32

						32			32
		32		32	64	64		32	64
32	64	64	128	128	128	128	256	256	256

Timsort: стек

При соблюдении инварианта:

- Размеры кусков, хранящихся в стеке растут: $r_i < r_{i+1}$
- Растут быстрее чисел Фибоначчи: $r_i + r_{i+1} < r_{i+2}$

  $r_2 > r_1$

  $r_3 > r_1 + r_2 > 2r_1$

  $r_4 > r_2 + r_3 > 3r_1$

  $r_5 > r_3 + r_4 > 5r_1$

  или (другое рассуждение):

  $r_{i+2} > r_i + r_{i+1} > 2r_i$

  т.е. скорость роста – экспоненциальная
- Следовательно, высота стека $O(\log n)$

Timsort: в чем ошибка?

				4	
			2	2	4
		3	3	3	5
	10	10	10	10	10
14	14	14	14	14	14

Мораль: После слияния r_2 и r_3 может потребоваться проверить условие $r'_2 + r'_3 < r'_4$.

Если этого не сделать: нарушается инвариант, рассуждение про ограниченность стека не верно. Реализации Timsort используют стек фиксированного размера (чтобы не тратить время на изменение размера массива) – в результате получаем выход за границу массива.

Timsort: реализация в Python 3.6.5

Algorithm 1: Timsort

Input: A sequence S to sort
Result: The sequence S is sorted into a single run

```
1 runs ← a run decomposition of S
2 R ← an empty stack
3 while runs ≠ ∅ do                          // main loop of Timsort
4     remove a run r from runs and push r onto R
5     merge_collapse(R)
6 if h ≠ 1 then
7     merge_force_collapse(R)
```

Algorithm 2: `merge_collapse`

Input: A stack of runs $\mathcal{R}$
Result: The invariant is established

```
1 while h > 1 do
2     if (h > 2 and r3 ⩽ r2 + r1) or (h > 3 and r4 ⩽ r3 + r2)
       then
3         if r3 < r1 then merge(R2, R3)
4         else merge(R1, R2)
5     else if r2 ⩽ r1 then merge(R1, R2)
6     else break
```

Timsort: слияние

Слияние всегда выполняется для соседних подмассивов.

Создаем временный массив, размер которого равен размеру меньшего из сливаемых подмассивов. Меньший из подмассивов копируем во временный массив.

Timsort: сложность

Разберем доказательство оценки сложности алгоритма Timsort, приведенное в статье

N. Auger, C. Nicaud, C. Pivoteau,
On the Worst-Case Complexity of TimSort.
`https://arxiv.org/pdf/1805.08612.pdf`

В анализе не пытаемся учитывать оптимизации слияния: считаем, что стоимость слияния двух `run` длины r и r' требует не более $r + r'$ сравнений.

Полученная оценка количества сравнений оказывается оптимальной: доказано, что количество сравнений для любого алгоритма сортировки оценивается снизу (с точностью до мультипликативной константы) такой же функцией. Следовательно, оптимизация слияния не меняет асимптотическую сложность.

Timsort: сложность

Теорема 1. Пусть $\mathcal{C}$ – множество массивов длины n, состоящих из runs длин $r_1, \ldots, r_\rho$. Введем функцию энтропии Шеннона

$$H(p_1, \ldots, p_\rho) = -\sum_{i=1}^{\rho} p_i \log p_i$$

и пусть $\mathcal{H} = H(r_1/n, ..., r_\rho/n)$. Время работы алгоритма Timsort на массивах из $\mathcal{C}$ равно $O(n + n\mathcal{H})$.

Теорема 2. $\mathcal{H} \leq \log \rho$, т.е. время работы алгоритма Timsort на массивах из $\mathcal{C}$ равно $O(n + n \log \rho)$, и следовательно, $O(n \log n)$.

Доказательство теоремы 2: надо показать, что $\mathcal{H} \leq \log \rho$.

Функция $f(x) = -x \ln(x)$ выпукла вверх на $\mathbb{R}_{>0}$, т.к. вторая производная отрицательна: $f''(x) = -1/x < 0$.

Неравенство Иенсена: если функция f выпукла вверх на D, то для любых $x_1, \ldots x_n \in D$ и $q_1, \ldots q_n$ таких, что $q_1 > 0, \ldots, q_n > 0$ и $q_1 + \ldots q_n = 1$, выполнено

$$\sum_{i=1}^{n} q_i f(x_i) \leq f\left(\sum_{i=1}^{n} q_i x_i\right).$$

Подставим $n = \rho$, $q_i = 1/n$, $x_i = p_i$. Поскольку $p_1 + \ldots + p_\rho = 1$, то

$$\sum_{i=1}^{\rho} \frac{1}{\rho} f(p_i) \leq f\left(\frac{1}{\rho} \sum_{i=1}^{\rho} p_i\right) = f\left(\frac{1}{\rho}\right).$$

Следовательно,

$$H(p_1, \ldots, p_\rho) = \sum_{i=1}^{\rho} f(p_i)/\ln(2) \leq \rho f(1/\rho)/\ln(2) = \log \rho.$$

Предложение 3. Для любого детерминированного алгоритма сортировки ***сравнением*** найдется массива из множества $\mathcal{C}$, на котором потребуется, по крайней мере, $n\mathcal{H} - 3n$ сравнений.

Идея доказательства такая же, как в доказательстве утверждения о том, что любой алгоритм сортировки сравнениями имеет сложность с худшем случае $\Omega(n \log n)$: посчитать количество разных массивов в классе $\mathcal{C}$, применить формулу Стирлинга.

С этой точки зрения Timsort оказывается асимптотически оптимальным алгоритмом! MergeSort и QuickSort таким свойством не обладают.

Timsort: реализация в Python 3.6.5

Algorithm 1: Timsort

Input: A sequence S to sort
Result: The sequence S is sorted into a single run

```
1 runs ← a run decomposition of S
2 R ← an empty stack
3 while runs ≠ ∅ do                          // main loop of Timsort
4     remove a run r from runs and push r onto R
5     merge_collapse(R)
6 if h ≠ 1 then
7     merge_force_collapse(R)
```

Algorithm 2: `merge_collapse`

Input: A stack of runs $\mathcal{R}$
Result: The invariant is established

```
1 while h > 1 do
2     if (h > 2 and r3 ⩽ r2 + r1) or (h > 3 and r4 ⩽ r3 + r2)
       then
3         if r3 < r1 then merge(R2, R3)
4         else merge(R1, R2)
5     else if r2 ⩽ r1 then merge(R1, R2)
6     else break
```

TIMSORT: ПЕРЕФОРМУЛИРОВКА

Algorithm 3: TIMSORT: translation of Alg. 1 and Alg. 2

Input: A sequence to S to sort
Result: The sequence S is sorted into a single run
Note: $h =$ height of the stack $\mathcal{R}$
$R_i =$ i^{th} top-most run in the stack $\mathcal{R}$
$r_i =$ the size of R_i.

```
1  runs ← the run decomposition of S
2  R ← an empty stack
3  while runs ≠ ∅ do                                   // main loop of TIMSORT
4      remove a run r from runs and push r onto R                    // #1
5      while true do
6          if h ⩾ 3 and r1 > r3 then merge(R2, R3)                   // #2
7          else if h ⩾ 2 and r1 ⩾ r2 then merge(R1, R2)              // #3
8          else if h ⩾ 3 and r1 + r2 ⩾ r3 then merge(R1, R2)         // #4
9          else if h ⩾ 4 and r2 + r3 ⩾ r4 then merge(R1, R2)         // #5
10         else break
11 while h ≠ 1 do merge(R1, R2)
```

Утверждение 4. Для любого массива алгоритмы 1 и 3 выполняют одну и ту же последовательность действий.

Замечание 5. Для анализа сложности Timsort достаточно оценить только сложность основного цикла:

- разбиение на `runs` требует $O(n)$ операций;
- цикл в конце алгоритма (`merge_force_collapse`) можно выполнить в ходе основного цикла, если добавить в конце массива фиктивный `run` размера $n + 1$.

Лемма 6. На любом шагу основного цикла Timsort, выполнено $r_i + r_{i+1} < r_{i+2}$ для всех $i \in \{3, \dots h - 2\}$.

Следствие 7. В основном цикле Timsort, перед тем как run помещается в стек, выполнено $r_i \leq 2^{(i+1-j)/2} r_j$ для всех $i \leq j \leq h$.

Доказательства технические, пропустим.

Выпишем последовательность случаев #1 ... #5, выполняемых в основном цикле алгоритма. Разобьем ее на части. Кусок, начинающийся с #1 и включающий все следующие #2 назовем *стартовым*. Кусок, начинающийся с #3, #4 или #5 и до следующего #1 назовем *завершающим*.

$$\underbrace{\#1\ \#2\ \#2\ \#2}_{\text{стартовый}}\quad\underbrace{\#3\ \#2\ \#5\ \#2\ \#4\ \#2}_{\text{завершающий}}\quad\underbrace{\#1\ \#2\ \#2\ \#2\ \#2\ \#2}_{\text{стартовый}}\quad\underbrace{\#5\ \#2\ \#3\ \#3\ \#4\ \#2}_{\text{завершающий}}$$

Замечание: слияния #4 и #5 не могут быть последними в завершающей последовательности! На следующей итерации основного цикла будет еще слияние.

Лемма 8. Стоимость всех операций `merge` выполняемых во время *стартовых* последовательностей равна $O(n)$.

Док-во. Покажем, что стартовая последовательность, начинающаяся с добавления в стеке `run` размера r использует не более γr сравнений, где $\gamma = 2\sum_{j=1}^{\infty} j/2^{j/2}$.
Если в стартовой последовательности k символов, то есть $k-1$ символов $\#2$, то общая стоимость равна

$$C = (k-1)r_1 + (k-1)r_2 + (k-2)r_3 + \ldots + r_k \leq \sum_{i=1}^{k}(k+1-i)r_i.$$

Для того, чтобы выполнился последний $\#2$ в стартовой последовательности должно быть выполнено $r > r_k$, и следовательно, $r \geq r_k \geq 2^{(k-1-i)/2} r_i$ для всех $i = 1, \ldots k$.
Следовательно,

$$C/r \leq \sum_{i=1}^{k}(k+1-i)2^{(i-k+1)/2} = 2\sum_{j=1}^{k} j2^{-j/2} < \gamma.$$

Для оценки сложности завершающих последовательностей воспользуемся амортизационным анализом (хотя оцениваем обычную сложность!).

Будем использовать 2 типа монет: $\diamondsuit$ и $\heartsuit$.

1. Когда элементы `run` *помещаются в стек* или когда они *опускаются в стеке во время выполнения завершающей последовательности*, каждому из них выдаем две $\diamondsuit$ монеты и одну $\heartsuit$ монету.
2. #2: сливаем r_2 и r_3. Каждый элемент из r_1 и r_2 платит по одной монете $\diamondsuit$. Монет хватает на оплату слияния r_2 и r_3, т.к. в этом случае $r_1 > r_3$.
3. #3: сливаем r_1 и r_2. Каждый элемент из r_1 платит две монеты $\diamondsuit$. Монет хватает на оплату слияния r_1 и r_2, т.к. в этом случае $r_1 \geq r_2$.
4. #4 и #5: сливаем r_1 и r_2. Каждый элемент из r_1 платит по одной монете $\diamondsuit$, а каждый элемент из r_2 платит по одной монете $\heartsuit$. Получаем ровно $r_1 + r_2$ монет на оплату слияния r_1 и r_2.

Лемма 9. Баланс монет обоих типов остается неотрицательным во время работы основного цикла Timsort.

При выполнении слияний #2 и #3 элементы, которые оплачивали слияние опускались в стеке – и мы возмещаем их расходы.

При выполнении слияний #4 и #5 потраченные ♢-монеты возмещаются сразу, а элементы, которые потратили ♡-монеты, оказываются на вершине стека, на следующей интерации цикла выполняется слияние, в котором они не тратят ♡-монеты, но при этом опускаются – и им возвращают ♡-монету. □

Следовательно, выданных монет хватает на оплату всех слияний.

Лемма 10. Высота стека после выполнения *стартовой* последовательности после добавления в стек r удовлетворяет неравенству

$$h \leq 4 + 2\log(n/r).$$

Док-во. Ни один из `run` $\bar{r}_3$, ... $\bar{r}_h$ не был затронут во время стартовой последовательности (#2 сливает r_2 и r_3, что дает r'_2). Поэтому для них выполнено $\bar{r}_3 \leq 2^{2-h/2}\,\bar{r}_h \leq 2^{2-h/2}n$.

Также во время выполнения стартовой последовательности не выполнялось слияние r_1, поэтому $r_1 = \bar{r}_1$, а после выполнения стартовой последовательности имеем $\bar{r}_1 \leq \bar{r}_3$. Отсюда получаем утверждение леммы. □

Следовательно, общее количество выданных монет равно

$$O\left(\sum_{i=1}^{\rho} r_i \log\left(\frac{n}{r_i}\right)\right) = O\left(-n\sum_{i=1}^{\rho}\frac{r_i}{n}\log\left(\frac{r_i}{n}\right)\right) = O\left(n\mathcal{H}\right).$$

Timsort: сложность

Разбиение на runs: $O(n)$.

Стартовые последовательности: $O(n)$.
Завершающие последовательности: $O(n\mathcal{H})$.
Итого, основной цикл: $O(n + n\mathcal{H})$.

В силу замечания 5, общая сложность Timsort: $O(n + n\mathcal{H})$.