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.

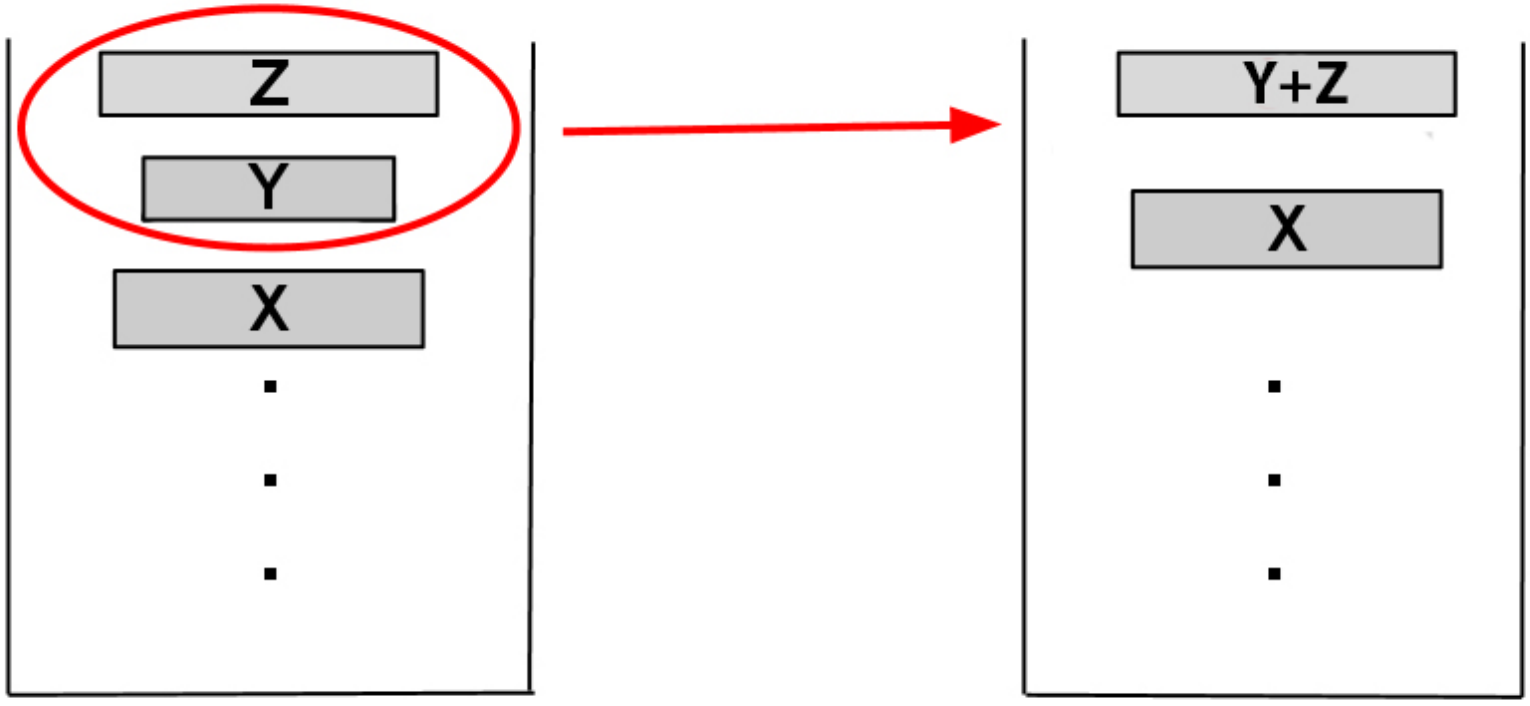
Пример: $N = 2112$
$\texttt{minrun} = 32$ – плохой $(2112 = 66 \times 32)$,
$\texttt{minrun} = 33$ – хороший $(2112 = 64 \times 33)$.

```
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: 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: 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: режим «галопа»

Still without loss of generality, assume A is the shorter run. In galloping mode, we first look for A[0] in B. We do this via "galloping" comparing A[0] in turn to B[0], B[1], B[3], B[7], ..., B[2**j - 1], ..., until finding the k such that B[2**(k-1) - 1] < A[0] <= B[2**k - 1]. This takes at most roughly lg(B) comparisons, and, unlike a straight binary search, favors finding the right spot early in B (more on that later).

After finding such a k, the region of uncertainty is reduced to 2**(k-1) - 1 consecutive elements, and a straight binary search requires exactly k-1 additional comparisons to nail it. Then we copy all the B's up to that point in one chunk, and then copy A[0]. Note that no matter where A[0] belongs in B, the combination of galloping + binary search finds it in no more than about 2*lg(B) comparisons.

If we did a straight binary search, we could find it in no more than ceiling(lg(B+1)) comparisons - but straight binary search takes that many comparisons no matter where A[0] belongs. Straight binary search thus loses to galloping unless the run is quite long, and we simply can't guess whether it is in advance.

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)
```

Timsort: сложность

Теорема. 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 $O(n + n\mathcal{H})$.

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

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

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

Функция $f : x \to -x \ln(x)$ вогнута на $\mathbb{R}_{>0}$, т.к. $f''(x) = -1/x < 0$.

Поскольку $p_i > 0$ и $p_1 + \ldots + p_\rho = 1$, то

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