Сравнение работы алгоритмов сортировки, реализованных на языке Python

22.06.2011

МГКН-1
Самунь Виктор

Оглавление

Общее описание тестирования

Тестирование алгоритмов сортировки заключается в измерении времени работы алгоритма и количества различных операций над массивом данных в зависимости от размера сортируемого массива.

Описание программного комплекса

Для проведения тестирования была написана программа, состоящая из следующих модулей:

1. `sorting.py` – главная программа, производит разбор аргументов, считывание и запись данных;
2. `sort.py` – модуль с алгоритмами сортировки и модульными тестами на алгоритмы;
3. `sort_report.py` –модуль с алгоритмами сортировки и модульными тестами на алгоритмы.

Модули `sort.py` и `sort_report.py` отличаются тем, что первый оптимизирован по времени работы, а второй позволяет получить информацию о количестве операций над массивом. Почему это было так сделано, будет пояснено позже.

Генерация входных данных и запуск `sorting.py` осуществлялись с помощью ранее написанных скриптов на Perl, которые использовались в (1). Мы посчитали бессмысленным занятием переносить этот функционал на Python, так как на время работы алгоритмов сортировки он не влияет. Кроме того, мы не генерировали входные данные. Мы использовали те же данные, что и в предыдущей работе, чтобы можно было корректно сравнить время работы на Perl и Python.

Алгоритмы

Для тестирования были реализованы следующие алгоритмы:

1. Встроенный алгоритм (просто вызов метода `sort`);
2. Быстрая сортировка;
3. Сортировка слиянием;
4. Пирамидальная сортировка;
5. Сортировка линейными вставками;
6. Сортировка бинарными вставками;
7. Сортировка Бэтчера;
8. Сортировка Шелла (8 модификаций);
9. Сортировка выбором;
10. «Пузырьковая» сортировка.

Подробный анализ алгоритмов сортировки приведён в (1).

Особенности реализации

Поясним для начала, зачем произведено разделение алгоритмов на два модуля.

В первом варианте все алгоритмы были реализованы в одном модуле, другой содержал два класса, внешне ведущих себя как массив и делегирующих вызовы в хранимый массив, но один подсчитывал количество различных операций, а второй – нет. В этом случае получалась хорошая объектная модель, но был обнаружен существенный недостаток. Один и тот же алгоритм сортировки, с одними и теми же данными, но для обычного массива и нашего класса, просто делегирующего вызовы, показывает различающееся на порядок время работы. Из-за этого все

полученные результаты по времени работы становятся неверными – слишком много накладных расходов.

Во втором варианте реализации данные передавались не через упаковку в специальный объект, умеющий подсчитывать количество вызовов, а через обычный массив, но в алгоритм встроен простой функционал по подсчёту количества операций – просто заведено 3 локальных переменных, у которых постепенно увеличивается значение. В этом случае время работы различалось до 10% для некоторых алгоритмов, что также достаточно много. Хотя у некоторых алгоритмов эта разница была несущественной – около 1%. Из-за того, что накладные расходы слишком разные (отличаются на порядок), от этого способа также было решено отказаться.

В итоге, был выбран способ, вразрез идущий с принципами ООП, трудно поддерживаемый, но показывающий достоверную информацию по сравнению времени работы. Способ заключается в отдельной реализации алгоритмов для сравнения по времени и сравнения числа операций.

Особенности запуска

Для начала мы опытным путём определили, что время работы различных алгоритмов на одном и том же массиве может существенно отличаться. Поэтому, как и в (1), входные данные были двух типов – массивы большого размера (использовались только в «быстрых» алгоритмах) и массивы маленького размера (использовались в каждом алгоритме).

Каждый тест запускался до 10 раз, и измерялось среднее время работы и погрешность измерений. Алгоритмы, которые работали слишком долго, запускались меньшее число раз (5 или 2), но при этом учитывалась погрешность – относительная погрешность не превышала 0.1%, что является хорошим результатом. Отметим, что вся логика по многократному запуску и измерению среднего времени работы, погрешности измерений реализована в главной программе – **`sorting.py`**. Время измерялось с помощью вызова **`time.time ()`**.

Тесты на измерение количества операций запускались по одному разу по понятным причинам – здесь погрешностей, связанных с измерением времени просто нет.

Полученные результаты

Вначале определим, что такое «быстрый» и «медленный» алгоритм. К «быстрым» алгоритмам мы относим те, которые имеют малое время работы и для которых был проведён полный объём запусков. Остальные алгоритмы относятся к «медленным».

Список «быстрых» алгоритмов:

1. Встроенный алгоритм (просто вызов метода `sort`);
2. Быстрая сортировка;
3. Сортировка слиянием;
4. Пирамидальная сортировка;
5. Сортировка бинарными вставками;
6. Сортировка Бэтчера;
7. Сортировка Шелла (модификации 1-7);

Список «медленных» алгоритмов:

1. Сортировка линейными вставками;
2. Сортировка Шелла (модификация 0);
3. Сортировка выбором;
4. «Пузырьковая» сортировка.

В файле `results.xlsx` приведены полные данные по времени работы. В этом же файле приведены данные из предыдущего отчёта.

Сравнение Perl vs Python

Как и в предыдущем отчёте, проранжируем алгоритмы сортировки каждого типа по времени работы на максимальном размере массива, для которого был произведён запуск. Кроме этого, сравним полученный результат с результатом из (1).

Таблица 1

Python		Perl		Perl / Python	
Алгоритм	Время работы, с	Алгоритм	Время работы, с	Алгоритм	Отношение
internal	2	internal	15	shell6	0,002178
qsort	16	qsort	34	shell4	0,002486
mergesort	41	shell6	44	shell3	0,002595
heap_sort	43	shell3	48	shell5	0,002887
batcher	144	shell4	51	shell1	0,006481
bin_insert	1218	mergesort	83	shell2	0,033822
shell7	1955	shell5	83	shell7	0,114578
shell1	14195	shell1	92	shell0	1,466511
shell3	18499	heap_sort	115	batcher	1,631944
shell6	20202	shell7	224	bin_insert	1,640394
shell4	20513	batcher	235	lin_insert	1,843583
shell2	23180	shell2	784	mergesort	2,02439
shell5	28752	bin_insert	1998	qsort	2,125
lin_insert	748	lin_insert	1379	select	2,227533
select	1046	shell0	1883	heap_sort	2,674419
shell0	1284	select	2330	bubble	3,029309
bubble	1433	bubble	4341	internal	7,5

Из таблицы можно заметить следующее. Во-первых, ранжирование алгоритмов в Perl и Python существенно отличается. На своих местах остались только лидер – внутренний вызов и аутсайдер – «пузырьковая» сортировка. Также отметим, что все модификации сортировки Шелла, кроме нулевой, на Perl работают на порядок (модификация 7) или даже на два (модификации 1-6) быстрее. Остальные алгоритмы на Python работают быстрее примерно в два раза. Внутренний алгоритм работает быстрее в 7 раз.

Анализ сложности

Снова, как и в предыдущем отчёте, приблизим полученные результаты функцией ax^b.

Таблица 2

Алгоритм	a	b
mergesort	9.7349e-6	1,07174
internal	3.7464e-7	1,0794
qsort	3.3572e-6	1,08086
heap_sort	8.8735e-6	1,08243
batcher	6.7483e-6	1,18701
shell7	1.4323e-8	1,8025
bubble	5.5295e-7	1,88238
shell6	1.4181e-8	1,96723
shell0	1.1923e-7	2,00664
lin_insert	6.135e-6	2,01693
shell2	7.3416e-9	2,02386
select	7.8234e-8	2,0246
shell4	5.4035e-9	2,03686
shell5	4.7487e-9	2,06955
shell3	2.3983e-9	2,08663
shell1	1.279e-9	2,11224
bin_insert	1.2478e-12	2,42705

Из таблицы можно заметить следующее. Во-первых, первые 5 алгоритмов из таблицы показывают почти линейный рост. Если посмотреть в таблицу 1, то можно заметить, что по скорости работы эти алгоритмы не менее чем на порядок превосходят все остальные.

Также можно отметить, что алгоритм бинарных вставок показывает наихудший результат по росту, даже хуже, чем линейные вставки, но «спасает» этот алгоритм то, что коэффициент у него (в среднем) более чем на 4 порядка меньше. Если исследовать причину такой работы более подробно, то можно отметить, что в случае рассмотрения небольших размеров массива, коэффициент b равен примерно 1.7 – результат, которого стоит ожидать. Для больших же размеров массива коэффициент возрастает до 2.4. Причина этому, скорее всего, в устройстве структуры данных, а именно, доступ к элементу по индексу осуществляется не за константу (по крайней мере, для больших массивов).

Кроме этого, заметим, что все сортировки Шелла имеют почти квадратичный рост вместо теоретических полутора. Причина этому та же самая – устройство структуры данных, которая «увеличивает» сложность.

Также интересует то, какой алгоритм сортировки используется во внутренней реализации. Несложно понять, что скорее всего, используется сортировка Хоара – так как их степени близки.

Анализ количества операций

Отметим, что раз мы использовали те же исходные данные и алгоритмы, что и в (1), то и результаты будут аналогичные. Таблицу приводить не будем. Совпадение результатов подтвердилось на практике, различие только было во внутреннем алгоритме сортировки. В Perl было произведено 120957 сравнений элементов, в Python же количество сравнений было меньше – 119891. Это объясняется внутренними оптимизациями реализации.

Заключение

Был написан программный комплекс, с помощью которого было произведено тестирование алгоритмов сортировки, реализованных на языке Python. Полученные результаты были сравнены с аналогичными результатами для языка Perl. Отмечено, какие алгоритмы сортировки работают быстрее, будучи реализованными на Perl и какие – на Python.

Кроме этого, были получены интересные результаты о возможном внутреннем устройстве структур данных и «плате за удобство» - использованию ООП.

Также был определён алгоритм сортировки, реализованный в стандартной библиотеке.

Список литературы

1. **Самунь, Виктор.** *Сравнение работы алгоритмов сортировки, реализованных на языке Perl.* УрГУ. Екатеринбург : б.н., 2007. стр. 21.