|  |
| --- |
| ТЕСТОВОЕ ЗАДАНИЕ |
| ГЕНЕРАЦИЯ И СОРТИРОВКА МОБИЛЬНЫХ НОМЕРОВ |
|  |
| Работу выполнил Поляков Дмитрий Сергеевич |
|  |

Репозиторий с исходным кодом: [github.com/johuex/tel\_numbers.git](https://github.com/johuex/tel_numbers.git).

Для разработки генератора и сортировщика использовался ЯП Python.

**Генератор**

Рассмотрим работу генератора. Изначально число генерировалось поразрядно (за исключением первых двух цифр – 8(9\*\*)\*\*\*-\*\*-\*\* ) от старшего к младшему по формуле:

**randint(0, 9) \* 10\*\*(8 - j),**

где j – порядок разряда в цикле.

Генерация таким способом занимала очень много времени (100 номеров за 180 секунд). В связи с этим генерацию было решено проводить не поразрядно. Было выяснено, что размер числа с максимальным значением номера равняется 32 байтам, а строка 60 байт (рисунок 1). Всего возможных вариантов номеров – 109, значит потребуется ~ 30517 Мб оперативной памяти. Для увеличения скорости генерации чисел и во избежание случая переполнения оперативной памяти диапазон номеров (9000000000, 9999999999) будет разбит на 27 частей. Каждая двадцать седьмая часть будет сгенерирована, перемешана и записана в один файл. Общее время генерации 109 номеров = 46 минут (рисунок 2). Объем каждой 27 части в оперативной памяти занимает 1130 (999`999`999/27\*32/1024\*\*2) Мб. Если ограничение по памяти 256 Мб, то тогда стоит разбить последовательность на 120 (30517 / 256) частей.

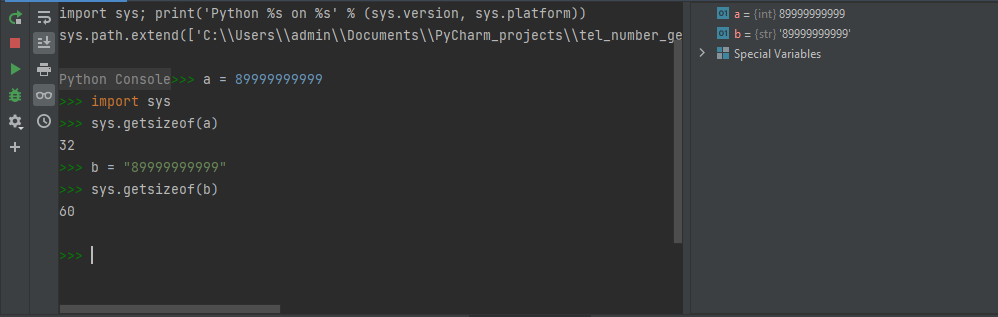


Рисунок 1 – Размеры Int и Str переменных

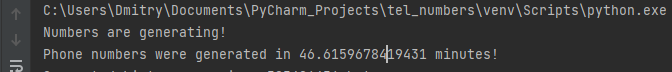


Рисунок 2 – Пример работы генератора

**Сортировка**

По причине нехватки оперативной памяти было принято решение сразу разрабатывать решение с ограничением по памяти (def sorter\_1). Был выбран алгоритм внешней сортировки – использует внешнюю память для хранения и записи.

Шаги алгоритма:

1. Разбиваем один большой файл на несколько (чем меньше оперативной памяти, тем больше будет более мелких файлов);
2. Сортируем содержимое каждого файла;
3. Закрываем исходный большой файл, дробные временные файлы не закрываем, открываем итоговый файл;
4. Считываем по одной строке из каждого временного файла;
5. Цикл:
   1. Выбираем наименьшую строку из всех файлов;
   2. Записываем ее в итоговый файл;
   3. Считываем новую строку из файла, в котором была наименьшая строка;
   4. Если в файле строк больше нет – закрываем его;
   5. Если все файлы закрыты, то работа цикла завершается, закрываем итоговый файл;
6. Удаляем все временные файлы.

Деление и сортировка 27 файлов занимает 43 минуты, внешняя сортировка – 32 минуты.

Если бы не ограничения по памяти, то для сортировки использовали бы следующий алгоритм (def sorter\_2): читаем построчно из файла номер телефона, далее переводим число из Str в Int. Далее номер помещается в массив, по которому проходимся до первого номера, меньшего, чем нынешний (сортировка номеров по возрастанию). Сортировка Int элементов происходит быстрее, чем Str элементов.

**Оптимизация генератора**

Для оптимизации и выявления узких мест в программе был проведен замер по времени отдельных шагов. В генераторе чисел для 1/27 шага всей последовательности требуется 108 секунд (рисунок 3). Как видно по рисунку узкими местами являются перемешивание массива и запись этой перемешанной части в файл.

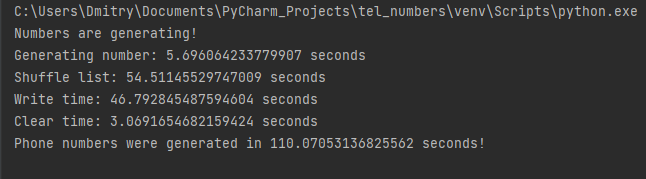


Рисунок 3 – Время работы 1/27 шага

Оптимизируем часть перемешивания – заменим функцию random.shuffle на свой алгоритм. Алгоритм меняет случайный элемент из первой половины списка с последним на момент итерации элементом из второй половины. Последовательность алгоритма:

* выбираем случайный элемент списка из первой половины;
* меняем его местами с (длина списка – итератор) элементом второй половины;
* цикл продолжается до тех пор, пока итератор не станет равен половине длины списка минус один.

Оптимизация записи в файл заключается в том, что теперь мы не построчно записываем в файл каждый элемент списка, а сразу весь список c переводом из Int в Str через операцию write с применением join. Общее время работы 1/27 цикла уменьшилось на 48% (рисунок 4).

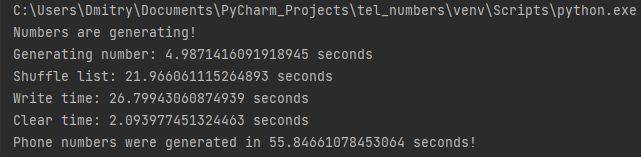


Рисунок 4 – Результат оптимизации работы генератора

**Оптимизация сортировки**

В сортировке также были произведены изменения. Чтение из файла теперь также происходит не построчно через цикл, а через чтение блоками с последующим переводом в список строк (*f\_in.read(12\*37037037).split('\n')*). Перевод списка строк в список целых чисел, сортировка и последующий обратный перевод с записью занимают гораздо больше времени, чем работа с обычными строками (без \n на конце). Время работы 1/27 цикла разбиения и сортировки временного мелкого файла занимает 71 секунду, вместо прежних 95 секунд (рисунок 5). Запись происходит через write и join. Время работы разделителя на мелкие сортированные файлы сократилось на 25% с 43 минут до 32 минут (1945 секунд) (рисунок 6).

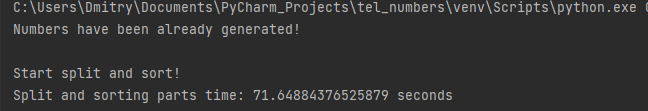


Рисунок 5 – 1/27 цикла разделение и сортировки на мелкие файлы

В цикле внешней сортировки запись также теперь производится не построчно, а через write и join. Время работы этой части сократилось на 21% с 32 минут до 25 (1520 секунд).

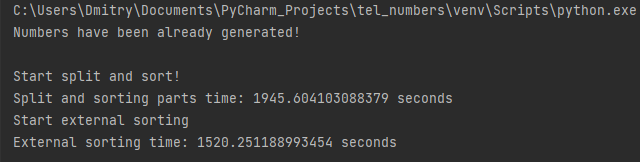


Рисунок 6 – Пример работы всего цикла внешней сортировки

**Итого** общее время работы всей системы после оптимизации занимает 82 минуты, вместо 121 (на 32% быстрее).