Национальный исследовательский университет “Высшая школа экономики”.

Факультет компьютерных наук. Программная инженерия.

Построение и анализ алгоритмов.

Домашнее задание №1 студента группы БПИ213 Абрамова Александра Сергеевича.

Разработанные материалы

При выполнении задания были разработаны следующие элементы:

1. Папка $sort$ содержит программу на языке C++, которая реализует 13 алгоритмов сортировки массива и измерение эффективности их работы.
   1.1. Выбор анализируемого алгоритма сортировки происходит при сборке программы путём указания соответствующей переменной препроцессора.
   1.2. Выбор метода измерения эффективности работы алгоритма также происходит при сборке программы путём указания переменной препроцессора $COUNT\_TIME$ для замера времени выполнения или $COUNT\_OPERATIONS$ для вычисления количества произведённых элементарных операций.
   1.3. При запуске программы происходит считывание тестируемого массива из файла, путь к которому указывается единственным аргументом командной строки, его сортировка и вывод необходимого результата: отсортированного массива при сборке в режиме $DEBUG$ или $TEST$, времени выполнения или количества элементарных операций.
   1.4. Элементарными операциями считались следующие инструкции:
      1) Получение значения переменной (чтение ячейки памяти)
      2) Установка значения переменной (запись ячейки памяти)
      3) Операции сравнения загруженных из памяти значений
      4) Арифметические операции (сложение, вычитание, умножение, деление) над загруженными из памяти значениями или константами
      5) Побитовые операции над загруженными из памяти значениями или константами
      6) Логические операции над загруженными из памяти значениями или константами
2. Папка $scripts$ содержит вспомогательные программы для генерации тестовых данных, сборки и запуска программ, а также для обработки выходных данных.
   2.1. Файл $config.js$ содержит настройки экспериментов:
      1) Список типов анализируемых массивов:

a) $RANDOM\_SMALL$ - массив из случайных чисел в интервале $[0;\ 5)$
b) $RANDOM\_BIG$ - массив из случайных чисел в интервале $[0;\ 4000)$
c) $ALMOST\_SORTED$ - почти отсортированный массив (среди каждых 50-ти элементов массива произведено 6 перестановок).
d) $BACKWARDS\_SORTED$ - массив, отсортированный в обратном порядке

2) Список групп тестов. Каждая группа представлена массивом тестов. Каждый тест описывается длиной массива.
3) Список анализируемых алгоритмов сортировки.
4) Количество запусков программы на каждом наборе тестовых данных при измерении времени выполнения для получения более точного результата.

2.2. Файл $gen.js$ содержит исходный код программы, генерирующей наборы тестовых данных. Для каждого типа массива генерируется эталонный массив длины 4100, из которого происходит копирование массивов меньшей длины при необходимости. Сгенерированные наборы тестовых данных размещаются в директории $tests$. Для каждого теста сохраняется набор входных данных $in.in$, корректный набор выходных данных $out.out$ и описание теста $description.txt$.

2.3. Файл $run.js$ содержит исходный код программы, производящей сборку и запуск алгоритмов:

2.3.1. При указании аргумента командной строки $TEST$ программа производит тестирование всех алгоритмов сортировки и выводит результат работы для каждой группы тестов. При этом сборка программы происходит с аргументами `-fsanitize=address,undefined -fno-sanitize-recover=all -Werror -Wsign-compare` для отлова ошибок во время выполнения.

2.3.2. При указании аргумента командной строки $DEBUG$ происходит аналогичное тестирование всех алгоритмов сортировки с выводом вердикта для каждого теста.

2.3.3. При указании аргумента командной строки $COUNT\_TIME$ происходит многократный запуск всех алгоритмов сортировки с целью измерения времени выполнения. Собранные данные записываются в файл $COUNT\_TIME.json$ директории $data$ папки $report$. При этом сборка программы происходит с флагом `-O2`, который позволяет компилятору применять необходимые оптимизации и измерять время выполнения более корректно.

2.3.4. При указании аргумента командной строки $COUNT\_ELEMENTARY\_OPERATIONS$ происходит запуск всех алгоритмов сортировки с целью измерения времени выполнения. Собранные данные записываются в файл $COUNT\_ELEMENTARY\_OPERATIONS.json$ директории $data$ папки $report$. При этом сборка программы происходит без флагов оптимизации во избежание удаления компилятором важных инструкций.

2.4. Файл $parse\_raw.js$ содержит исходный код программы, обрабатывающей файлы $COUNT\_TIME.json$ и $COUNT\_ELEMENTARY\_OPERATIONS.json$. Программа генерирует файл $raw.xlsx$, содержащий все полученные результаты, а также $data.json$, содержащий полученные результаты после усреднения результатов измерения времени.

2.5. Файл $parse.js$ содержит исходный код программы, которая по файлу $data.json$ генерирует таблицу $results.xlsx$, содержащую информацию о среднем времени выполнения и количестве элементарных операций на каждом тесте каждой группы для каждого алгоритма сортировки, сгруппированную по типу тестируемого массива.

3. Файл $package.json$ содержит описания необходимых для удобной работы настроек: зависимость от библиотеки $exceljs$, которая использовалась для удобной генерации xlsx-файлов, а также ряд “скриптов” для запуска соответствующих вспомогательных программ с различными аргументами командной строки. В частности, скрипт $analyze$ проводит эксперимент целиком: запускает программы измерения времени работы и количества элементарных операций всех алгоритмов на всех наборах тестовых данных, а также программы $parse\_raw$ и $parse$ для обработки полученных данных.

Эксперименты и анализ

Для получения результатов скрипт $analyze$ был дважды запущен на разных системах под управлением $macOS$ и $WSL\ Debian$. Результаты сохранены в директориях mac и wsl папки $report$ соответственно. Файлы $SystemInformation.txt$ содержат описания окружения при проведения каждого из экспериментов.

Для построения графиков следует открыть файл $graphs.html$ (или $graphsLogScale.html$ для использования логарифмической шкалы) директории $report$ в любом браузере и загрузить файл $data.json$ соответствующего эксперимента.

В качестве основных результатов для анализа были выбраны данные, полученные на ОС $WSL\ Debian$. По графикам можно отметить следующее:

1. Сортировка пузырьком показывает худшие результаты среди анализируемых алгоритмов как на маленьких, так и на больших данных, что подтверждается подсчётом количества элементарных операций. Заметим, что условия Айверсона не позволяют получить значительного улучшения эффективности алгоритма на случайных данных, но улучшают производительность алгоритма при вводе почти отсортированного массива. Также отметим, что наибольшее количество элементарных операций происходит при вводе обратно отсортированного массива, что ожидаемо, но интересно, что измерение времени показывает немного другой результат на обеих ОС. Скорее всего, это вызвано архитектурными оптимизациями, которые применяются к программе компилятором во время сборки и процессором во время выполнения.
2. Сортировка выбором уверенно занимает второе с конца место: как на маленьких, так и на больших входных данных этот алгоритм значительно лучше сортировки пузырьком, но хуже всех остальных. Лишь на очень маленьких данных (длины менее 200) сортировка слиянием показывает чуть более плохой результат. Тем не менее интересно, что на $macOS$ на больших входных данных сортировка выбором оказывается худшей, что не подтверждается измерением количества элементарных операций. Вероятно, такой результат также вызван архитектурными особенностями $macOS$. Ожидаемо, сортировка выбором показывает приблизительно одинаковые результаты на массивах всех типов.
3. Далее идут два вида сортировки вставками. Интересно, что сортировка бинарными вставками не оказывается эффективнее сортировки простыми вставками: по количеству элементарных операций первый алгоритм незначительно проигрывает, но на практике разница во времени выполнения оказывается ещё более заметна, особенно на $macOS$. Также хочется отметить, что обе вариации сортировки вставками показывают себя значительно лучше среднего на почти отсортированных входных данных и заметно хуже среднего на данных, отсортированных в обратном порядке. Более того, на почти отсортированных данных сортировка простыми вставками оказывается эффективнее всех нелинейных алгоритмов сортировки.
4. Далее на графиках явно выделяется группа из четырёх алгоритмов: пирамидальная сортировка, сортировка слиянием и две вариации сортировки Шелла.
   4.1. С точки зрения количества элементарных операций явно видно, что пирамидальная сортировка показывает худший результат на случайных данных большого размера, а сортировка слиянием - лучший. Сортировка Шелла с последовательностью Циура, ожидаемо, заметно обходит сортировку Шелла с последовательностью Шелла.

4.2. Но замеры времени показывают немного другой результат: на маленьких данных сортировка слиянием оказывается худшим алгоритмом сортировки данных (скорее всего, из-за выделения дополнительной памяти), за ней следует пирамидальная сортировка, а далее - сортировки Шелла, причём от выбора последовательности результат зависит незаметно мало. Более того, все алгоритмы оказываются медленнее или лишь незначительно быстрее сортировки вставками. На большой длине массива результат иной: сортировка слиянием незначительно превосходит остальные алгоритмы, различия во времени выполнения которых практически нет.

4.3. Интересно, что график как времени выполнения, так и количества элементарных операций для сортировки Шелла с последовательностью Шелла не ровный, а возрастает скачкообразно, что вызвано устройством алгоритма: при увеличении длины массива количество итераций внешнего цикла увеличивается неравномерно, что и вызывает скачки. Ярко выраженный последний скачок происходит при последнем увеличении длины с 4000 до 4100 - добавляется ещё одна итерация, так как $log_2(4000) < 12$, а $log_2(4100) > 12$

4.4. Для рассматриваемых алгоритмов сортировки нет ярко выраженных отличий в выполнении на разных ОС: результат, полученный на устройстве под управлением $macOS$ примерно совпадает с описанным, только графики получились чуть более плавными и чуть более явно “отсортировались” по эффективности, при чём пирамидальная сортировка оказалась наиболее эффективной. Тем не менее отклонения незначительны.

4.5. Ожидаемо, для сортировки слиянием и пирамидальной сортировки тип массива не оказывает большого влияния на эффективность, но интересно, что для сортировки Шелла независимо от выбранной последовательности эффективность оказывается худшей при сортировке случайного массива с большими числами. И это подтверждается подсчётом количества элементарных операций.

5. Далее выделяется алгоритм быстрой сортировки с выбором первого элемента в качестве опорного. На маленьких данных среди нелинейных сортировок алгоритм оказывается хуже лишь сортировки простыми вставками, но уже на длине около 400 оказывается эффективнее всех остальных алгоритмов, что подтверждается и подсчётом количества элементарных операций. Заметим, что алгоритм работает значительно хуже на обратно отсортированном массиве, что ожидаемо: при выборе первого элемента как опорного на обратно отсортированных данных алгоритм “деградирует” до асимптотической сложности $O(n^2)$.

6. Ожидаемо, линейные сортировки оказываются наиболее эффективными на больших данных.
   6.1. Превосходство проявляется лишь на массивах длины более 400, а на массиве длины 50 сортировка подсчётом наоборот показывает себя не очень хорошо.
   6.2. Стоит заметить, что сортировка подсчётом показывает более хорошие результаты при обработке случайного массива с числами в интервале $[0; 5)$, в то время как цифровая сортировка превосходит при вводе чисел в интервале $[0; 4000)$, что ожидаемо - сортировка подсчётом требует дополнительных операций при большом "разбросе" значений массива, а цифровая сортировка работает "вхолостую" при маленьком разбросе значений. Тем не менее хочется отметить, что при увеличении длины массива эффективность сортировки подсчётом возрастает и она приближается к цифровой сортировке и в случае массивов с элементами в интервале $[0; 4000)$, а по количеству элементарных операций сортировка подсчётом даже превосходит цифровую сортировку при длине массива более 2000 (хотя на практике эта граница больше из-за особенностей работы компьютера). Интересно, что на $macOS$ эта граница действительно лишь немного выше 2000.
   6.3. По указанным ранее причинам, "худший случай" цифровой сортировки - случайный массив с элементами в интервале $[0; 5)$, хотя количество элементарных операций это не подтверждает, а указывает на то, что цифровая сортировка работает на любых данных одинаково эффективно.
   6.4. Для сортировки подсчётом результат "лучшего случая" подтверждается как измеренным временем, так и количеством выполненных элементарных операций: алгоритм работает наиболее эффективно на случайном массиве с элементами в интервале $[0; 5)$ и наименее эффективно при наличии в массиве больших элементов (случайный массив с элементами до 4000 и обратно отсортированный массив). Интересно, что для почти отсортированного массива эффективность сортировки монотонно убывает, что понятно: по построению наборов тестовых данных, чем больше длина почти отсортированного массива, тем более большие элементы в нём встречаются, что оказывает негативное влияние на производительность.
7. По полученным графикам можно предположить асимптотическую сложность каждого из анализируемых алгоритмов сортировки в лучшем, худшем и среднем случаях.

Алгоритм сортировки	Сложность в лучшем случае	Сложность в среднем случае	Сложность в худшем случае
Пузырьком	$O(n^2)$	$O(n^2)$	$O(n^2)$

Пузырьком с условием Айверсона 1	$O(n)$	$O(n^2)$	$O(n^2)$
Пузырьком с условием Айверсона 1+2	$O(n)$	$O(n^2)$	$O(n^2)$
Выбором	$O(n^2)$	$O(n^2)$	$O(n^2)$
Простыми вставками	$O(n)$	$O(n^2)$	$O(n^2)$
Бинарными вставками	$O(n)$	$O(n^2)$	$O(n^2)$
Слиянием	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
Пирамидальная	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
Быстрая с первым опорным	$O(nlogn)$	$O(nlogn)$	$O(n^2)$
Шелла с последовательностью Шелла	$O(nlogn)$	$O(nlogn)$	$O(n^2)$
Шелла с последовательностью Циура	$O(n)$	$O(nlogn)$	$O(n^2)$
Подсчётом (устойчивая)	$O(n + k)$	$O(n + k)$	$O(n + k)$
Цифровая	$O(nk)$	$O(nk)$	$O(nk)$

Вывод

1. При решении алгоритмических задач следует выбирать подходящий алгоритм сортировки:
   1.1. При небольшой длине входных данных следует отдавать предпочтение алгоритму сортировки простыми вставками, который показывает себя лучше остальных при вводе случайного или почти отсортированного массива, или алгоритму сортировки Шелла, если в качестве входных данных ожидается обратно отсортированный массив.
   1.2. Если длина вводимого массива превышает 400, имеет смысл, по возможности, использовать один из линейных алгоритмов сортировки, причём при небольшом диапазоне сортируемых значений эффективнее оказывается использовать сортировку подсчётом, иначе - цифровую сортировку.

1.3. При невозможности использования линейного алгоритма следует провести детальный анализ ожидаемых входных данных:
    1.3.1. Если ожидаются случайные данные из достаточно большого диапазона, то быстрая сортировка окажется лучшим решением.
    1.3.2. Если ожидаются случайные данные из небольшого диапазона, то сортировка Шелла с последовательностью Циура проявляет себя лучше остальных, а быстрая сортировка заметно теряет производительность из-за большого количества повторяющихся значений и низкой вероятности разделить данные примерно пополам выбором первого элемента как опорного.
    1.3.3. Если ожидаются почти отсортированные данные, то должен быть использован алгоритм сортировки простыми вставками, который имеет линейную сложность в этом случае и значительно превосходит все остальные алгоритмы, лишь немного отставая от линейных сортировок.
    1.3.4. Если ожидаемая “природа” данных неизвестна, то нужно использовать алгоритм, наименее зависящий от типа вводимого массива - алгоритм сортировки Шелла с последовательностью Циура, который оказывается лучшим при работе с обратно отсортированным массивом и с массивом, который содержит случайные данные в небольшом диапазоне, и делит второе место для остальных типов входных данных.
2. Для оценки эффективности алгоритма целесообразно использовать измерение времени его выполнения на различных наборах входных данных. Несмотря на то, что на результаты такого эксперимента могут оказывать влияние внешние факторы, большое количество измерений позволяет “сгладить выбросы”, в то время как измерение количества элементарных операций хотя и не содержит “выбросов”, но и не отражает все особенности выполнения алгоритма и не учитывает возможности применения оптимизаций во время компиляции и выполнения, которые могут заметно повлиять на результат. При оценке эффективности важно также учитывать особенности различных операционных систем и основывать выбор на требованиях целевой платформы, что не позволяет делать теоретическая оценка времени работы - подсчёт количества элементарных операций.
3. $MacOS$ позволяет минимизировать влияние внешних факторов на выполнение алгоритма (графики получились более гладкие), но в то же время усложняет его анализ, так как некоторые полученные результаты по архитектурным причинам имеют значительные отклонения от теоретических ожиданий, что подтверждает важность оценки эффективности алгоритма по времени его

выполнения: измерение количества элементарных операций не позволяет учесть особенности устройства различных компьютеров.