Задание SOR
Сортировка массива

Задача сортировки (упорядочивания) массива решается далеко не единственным способом. Известно много базовых схем, и в каждой из них возможны вариации, которые отличаются не только вычислительной сложностью, но и просто временем работы — константами, зависящими от количества сравнений и копирований элементов.

В данной работе требуется запрограммировать некоторый набор алгоритмов сортировки (вариаций) и произвести их наглядное сравнение. Для этого необходимо измерить среднее время работы каждого алгоритма на случайных массивах определённой длины. Из этих времён составить таблицу, показывающую зависимость среднего времени работы каждого алгоритма от размера массива, а по таблице построить графики в Excel. Первый столбец таблицы должен содержать размер массива, остальные — время работы, по столбцу на алгоритм, столбцы должны быть озаглавлены. Таблица должна быть сохранена в обычный текстовый файл формата CSV (Comma Separated Values), который в то же время легко открывается в Excel.

Для соблюдения принципа модульности все сортировки должны быть оформлены в виде отдельной библиотеки, а замер времени с созданием отчёта — в виде консольного приложения-профайлера. Чтобы обеспечить лучшую взаимозаменяемость библиотек, в этой задаче будем использовать динамические библиотеки (DLL на Windows, SO на Linux) с динамическим связыванием (действующие как «плагины»). При таком подходе ваше приложение-профайлер будет работать с любой другой библиотекой сортировок и наоборот.

Тестовое приложение не только занимается сбором статистики, но и проверяет правильность работы вызываемых алгоритмов (в отладочной конфигурации). Для этого необходимо после каждой сортировки сравнивать результат с эталонным, полученным при помощи библиотечной функции `qsort()`.

Ваша задача не просто реализовать заданные алгоритмы, но постараться сделать это наиболее быстрым образом, оптимизируя вычисления. Возможно, стоит поискать и другие более быстрые модификации алгоритмов, не указанные в задании.

Комментарии к задаче

Общность В библиотеке все алгоритмы должны быть реализованы в виде `static`-функций с одинаковым, фиксированным для всех прототипом, соответствующим приведённому указателю на функцию:

```
void (*sort_func_t)(int array[], int size);
```

Динамическая библиотека должна экспортировать только одну функцию:

```
__declspec(dllexport) sort_info_t* GetSortList(int* count);
```

которая возвращает указатель на первый элемент массива и количество элементов в нём. Каждый элемент этого массива соответствует одному реализованному алгоритму сортировки:

```
// Описание одной вариации сортировки
typedef struct
{
  sort_func_t sort;              // Функция сортировки
  sort_family_t family;          // Базовое семейство
  sort_complexity_t complexity; // Вычислительная сложность
  char const* name;              // Имя, описывающее вариацию
} sort_info_t;

// Семейства сортировок
typedef enum
{
  // (Не определено)
  SORT_NA = -1,
  // Квадратичные
  SORT_SELECTION, SORT_INSERTION, SORT_BUBBLE,
  // Субквадратичные
  SORT_SHELL,
  // Квазилинейные
  SORT_COMB, SORT_MERGE, SORT_QUICK, SORT_HEAP,
  // Линейные
  SORT_COUNT, SORT_RADIX, SORT_BUCKET,
} sort_family_t;

// Вычислительная сложность
typedef enum
{
  SORT_QUADRATIC,
```

```
    SORT_SUBQUADRATIC,
    SORT_QUASILINEAR,
    SORT_LINEAR
} sort_complexity_t;
```

В тестовом приложении вы будете использовать только поля `sort` и `name`, остальное будет учитываться преподавателем при проверке.

Локализация Формат CSV в оригинальном своём исполнении, как это следует из названия, содержит ячейки таблицы, разделённые запятыми. В русских настройках Excel запятая используется для разделения целой и дробной части вещественных чисел, поэтому элементы строки таблицы разделяются точкой с запятой. Грамотно воспользуйтесь `locale.h`, чтобы сохранить в файл нужные разделители.

Усреднение Обратите внимание, что нас интересует среднее время, т.е. необходимо усреднять по нескольким случайным массивам одного размера (напр., `REPEAT_COUNT` равный 5, 10, или 100). В то же время для сравнения разных сортировок необходимо использовать одинаковые массивы. Вспомните, что функция `rand()` генерирует последовательность псевдослучайных чисел, которая может быть воспроизведена сколько угодно раз. Для сброса генератора в начало последовательности используется функция `srand(r0)`.

Детализация Чтобы понять зависимость времени работы алгоритма от размера массива и получить плавный график, нельзя ограничиваться одним, двумя, пятью значениями N. Размер массива следует менять от `MIN_SIZE` до `MAX_SIZE` с шагом `SIZE_STEP`, значения параметров подобрать в зависимости от интересующего вас диапазона и от скорости работы алгоритма (например, от 1 до 100 с шагом 1, от 10 до 10000 с шагом 10 и т.п.).

Профилирование Мы хотим получить как можно более точное время работы алгоритма, поэтому при измерении времени следует использовать подходящие функции стандартной библиотеки и операционной системы. При недостаточной точности необходимо прибегнуть к дополнительным способам уменьшения погрешности (повторное выполнение

алгоритма). Разумеется, лишние отладочные проверки тоже не должны мешать правильным измерениям, поэтому профилирование принято проводить не в отладочной, а в финальной конфигурации проекта, с включенными оптимизациями компилятора.

План решения

Как полагается в разработке программного обеспечения, инструментарий определяется вариантами его использования. Следовательно, начинать разработку следует с написания небольшого тестового приложения.

1. Для начала создайте два проекта: один будет собираться в динамическую библиотеку, другой — в исполняемый файл. Проставьте зависимость второго проекта от первого, чтобы изменения в библиотеке вели к её своевременной перекомпиляции во время запуска тестового приложения.

2. Далее, в библиотеке добавьте заголовочный файл нужного формата и файл с исходным кодом, в котором реализуйте функцию `GetSortList()` и добавьте одну функцию сортировки, имеющую соответствующий прототип. Эта функция, например, может содержать вызов функции `qsort()`.

3. Далее, попробуйте загрузить библиотеку с помощью функции `LoadLibrary()` и после этого воспользоваться функцией `GetProcAddress` для доступа к адресу функции `GetSortList()`. Убедитесь, что возвращаются корректные значения, функция вызывается и возвращает то, что ожидается.

4. После этого, можно реализовать верификацию алгоритмов и измерение времени с учетом комментариев выше. Начать рекомендуется с кода генерации случайных массивов. Затем стоит реализовать последовательный вызов алгоритмов сортировки, полученных из `GetSortList()`, контролируя, что в одном измерении алгоритмы запускаются на одних и тех же данных. Далее можно добавить проверку того, что функция действительно сортирует введенный

массив и завершить всё реализацией подсчета времени и вывода таблицы.

5. В итоге, после того, как вы реализуете и протестируете весь сценарий использования, можно будет очень легко добавлять новые алгоритмы, убеждаться в их корректности и производить замеры времени.

Варианты

Вариант SOR–1 (Выбором и пузырьком). В рамках общего условия задачи реализовать алгоритмы сортировки:

1. Выбором (Selection sort)
   - со сдвигом (стабильная),
   - с обменом,
   - ⋆ квадратичный выбор ($\sqrt{N}$ блоков по $\sqrt{N}$ элементов).
2. Пузырьком (Bubble sort)
   - просто N раз по N,
   - остановка каждый раз на 1 раньше,
   - остановка если не было обменов,
   - шейкером (Cocktail sort),
   - расчёской / гребнем (Comb sort) с фактором 2 и 1.3.

Рекомендованные источники: Вирт [1, §2.2.2–2.2.3], Кнут [3, §5.2.2–5.2.3], Макконнелл [4, §3.2].

Вариант SOR–2 (Вставками). В рамках общего условия задачи реализовать алгоритмы сортировки вставками (Insertion sort):

- простые вставки с поиском слева направо,
- простые вставки с поиском справа налево,
- бинарные вставки,
- попарные вставки с поиском сразу двух элементов,
- ⋆ двухпутевые вставки с дополнительной памятью.

Рекомендованные источники: Вирт [1, §2.2.1], Кормен [2, §1.1], Кнут [3, §5.2.1], Макконнелл [4, §3.1].

Вариант SOR–3 (Шелла). В рамках общего условия задачи реализовать алгоритмы сортировки Шелла (Shellsort) с убывающими шагами вида:

- $\Delta_k = \lfloor \Delta_{k-1}/2 \rfloor ; \quad \Delta_0 = N,$
- $\Delta_k = \lfloor 5\Delta_{k-1}/11 \rfloor ; \quad \Delta_0 = N,$
- $\Delta_k \in \{2^n - 1\}_{n>0},$
- $\Delta_k \in \{4^n + 3 * 2^{n-1} + 1\}_{n>0} \cup 1,$
- $\Delta_k \in \{2^n \cdot 3^m\}_{n,m>0},$
- эмпирическая $\ldots, 701, 301, 132, 57, 23, 10, 4, 1$ начинающаяся как $\Delta_{k-1} = 2.25\Delta_k$

Рекомендованные источники: Вирт [1, §2.3.1], Кнут [3, §5.2.1], Макконнелл [4, §3.3].

Вариант SOR–4 (Слиянием). В рамках общего условия задачи реализовать алгоритмы сортировки слиянием (Merge sort):

- сверху вниз,
- снизу вверх,
- естественная,
- разбиение на три части,
- ⋆ без дополнительной памяти.

Рекомендованные источники: Вирт [1, §2.4.1–2.4.3], Кормен [2, §1.3], Кнут [3, §5.2.4], Макконнелл [4, §3.6].

Вариант SOR–5 (Быстрая и кучей). В рамках общего условия задачи реализовать алгоритмы сортировки:

1. Быстрая (Quicksort)
   - перекидывая слева направо (Ломуто),
   - классическое разбиение с двух концов (Хоара),
   - при маленьком размере просто останавливаться, потом целиком вставками,
   - выбор pivot случайный,
   - выбор pivot как медианы из трех.
2. Пирамидальная / кучей (Heapsort)
   - просеивание сверху вниз,
   - только с одним сравнением в узле («снизу вверх»).

Рекомендованные источники: Вирт [1, §2.3.2–2.3.3], Кормен [2, §7.1–7.5, §8.1–8.4], Кнут [3, §5.2.2–5.2.3], Макконнелл [4, §3.5, §3.7].

Вариант SOR–6 (Линейные). В рамках общего условия задачи реализовать алгоритмы сортировки:

1. Цифровая (Radix sort)
   - десятичная система,
   - двоичная система,
   - байтовая вариация (по основанию 256).
2. Черпаком (Bucket sort).

Рекомендованные источники: Кормен [2, §9.2–9.4], Кнут [3, §5.2.5], Макконнелл [4, §3.4].

Литература

[1] Вирт Н. Алгоритмы и структуры данных. Новая версия для Оберона. — М. : ДМК Пресс, 2016. — 272 с.

[2] Алгоритмы. Построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. — М. : Вильямс, 2016. — 1328 с.

[3] Кнут Д. Э. Искусство программирования, том 3. Сортировка и поиск. — М. : Вильямс, 2014. — 832 с.

[4] Макконнелл Дж. Анализ алгоритмов. Активный обучающий подход. — М. : Техносфера, 2009. — 416 с.

[5] Макконнелл С. Совершенный код. — М. : Русская редакция, 2015. — 896 с.

[6] Подбельский В., Фомин С. Курс программирования на языке Си. — М. : ДМК Пресс, 2015. — 384 с.

[7] Портал по функциям языков C и C++ // http://www.cplusplus.com/reference. — 2000. — Accessed: 03.03.2016.