ЗАДАНИЕ №4

ЧАСТЬ 1

Цель: В рамках первой части задания я проведу исследование некоторых функций языка C, позволяющих измерять определенные интервалы времени, и сравню их между собой.

Исследование:

Таблица 1. Функция gettimeofday

Величина задержки, мс	Результат измерений, мс	Количество повторов	RSE, %	Абсолютная ошибка, мс	Относительная ошибка, %
1000	1000.34	100	0	0.34	0.03
100	100.24	100	0.01	0.24	0.24
50	50.20	100	0.01	0.20	0.40
10	10.17	100	0.05	0.17	1.69

Функция gettimeofday получает текущее время, выраженное в секундах и микросекундах с 1 января 1970 года. По результатам измерений можно сделать вывод, что чем более маленький интервал времени мы меряем , тем большую относительную стандартную ошибку мы получаем. Но эта ошибка довольно мала даже при измерении интервалов порядка 50 мс (всего лишь 0.01%). Заметим также, что при уменьшении длительности измеряемого интервала времени, уменьшается и абсолютная погрешность.

Таблица 2. Функция clock_gettime

Величина задержки, мс	Результат измерений, мс	Количество повторов	RSE, %	Абсолютная ошибка, мс	Относительная ошибка, %
1000	1000.31	100	0	0.31	0.03
100	100.23	100	0.01	0.23	0.23

50	50.22	100	0.01	0.22	0.44
10	10.18	100	0.05	0.18	1.77

Функция clock_gettime с параметром CLOCK_MONOTONIC_RAW предоставляет время прошедшее с неопределенного момента в прошлом (будь то начало эпохи или время запуска системы). Ее результаты буквально такие же, как и у функции gettimeofday, что, в принципе, неудивительно, потому что принципы работы функций очень похожи.

Таблица 3. Функция clock

Величина задержки, мс	Результат измерений, мс	Количество повторов	RSE, %	Абсолютная ошибка, мс	Относительная ошибка, %
1000	1000.05	100	0	0.05	0.01
100	100.05	100	0	0.05	0.05
50	50.05	100	0	0.05	0.1
10	10.05	100	0.01	0.05	0.48

Базовым средством для работы с часами процессорного времени является функция clock(). Она возвращает в качестве результата процессорное время, затраченное процессом с некоего момента, зависящего от реализации и связанного только с его запуском. Проблема замера времени этой функции была в том, что nanosleep приостанавливает работу потока и процессорное время в этот момент не считается. Соответственно приходилось вручную добавлять к результатам измерения интервала сам интервал в тиках. Посмотрим на таблицу и увидим, что абсолютная погрешность везде одинаковая. Возможно, это связано с тем, что компьютеру легче считать тики, чем секунды. Результаты измерения очень точные даже при измерения интервалов времени порядка 10 мс.

Таблица 4. Функция __rdtsc

Величина задержки, мс	Результат измерений, мс	Количество повторов	RSE, %	Абсолютная ошибка, мс	Относительная ошибка, %
1000	997.25	100	0	2.75	0.28
100	99.98	100	0.01	0.02	0.02
50	50.06	100	0.01	0.06	0.11
10	10.14	100	0.05	0.14	1.43

Это самая неудобная для измерения интервала времени функция, потому что она предоставляет время работы функции в тактах процессора. Чтобы перевести их в секунды я использовал номинальную тактовую частоту своего процессора. Да, результаты нельзя считать точными (что даже видно по измерениям) из-за того, что номинальная тактовая частота не равна реальной тактовой частоте во время выполнения моей программы. Но тем не менее даже несмотря на это, на довольно маленьких интервалах времени, функция __rdtsc работает быстрее функций gettimeofday и clock_gettime.

Вывод: В теории самый точный метод измерения времени — это измерение с помощью функции __rdtsc, но этот способ является самым ненаглядным и неудобным, потому что мы получаем результат измерения в тактах процессора, которые сложно переводить в секунды (но при этом самому процессору считать в них проще, чем в секундах). На практике же самый точный способ с помощью функции clock. Способы с помощью функций gettimeofday и clock_gettime являются довольно неплохими, но уступают в точности предыдущему способу.

ЧАСТЬ 2

Цель: Вторая часть исследования направлена на сравнение времени работы сортировки выбором, реализованной тремя разными способами (операция

индексации, адресная арифметика и через указатели) + сравнить время работы сортировки на уже отсортированном массиве и на массиве общего вида

Исследование:

Для проведения исследования было реализовано 6 программ (сортировка массива общего вида, написанная с помощью индексации; сортировка массива общего вида, написанная с помощью адресной арифметики; сортировка массива общего вида, написанная с помощью указателей; сортировка упорядоченного массива, написанная с помощью индексации; сортировка упорядоченного массива, написанная с помощью адресной арифметики; сортировка упорядоченного массива, написанная с помощью указателей), каждую из которых я компилировал дважды (первый раз с O0-уровнем оптимизации, второй раз с O2-уровнем оптимизации). Программы сами генерировали массивы для дальнейшей сортировки.

Также были написаны следующие скрипты:

1. Скрипт *build_apps.sh* компилирует каждую из программ дважды. На выходе получается 12 исполняемых файлов (расположенных в каталоге ./*apps*), которые передаются скрипту *create_data.sh*

2. Скрипт *create_data.sh* запускает эти 12 исполняемых файлов (в качестве аргументов скрипт принимает шаг исследования STEP, максимальный размер массива MAXSIZE и количество замеров для сортировки каждым способом EXP) и записывает результаты в 12 текстовых файлов, располагающихся в каталоге ./*measure*, предварительно очищая его от прошлых измерений (как мне кажется, нет необходимости хранить прошлые замеры, потому что условия в которых проходит эксперимент могут поменяться (например, производительность процессора упала из-за открытия дополнительных окон), соответственно могут и измениться новые результаты)

3. Скрипт *make_preproc.sh* получает на вход файлы, обработанные предыдущим скриптом, и запускает питоновский скрипт *make_preproc.py,* который обрабатывает всю информацию из файлов (находит максимум, минимум, среднее, RSE и т.д) и записывает эту информацию в одноименные файлы,

расположенные в каталоге *measure/reses*. По информации, записанной в этих файлах, я строил все таблицы.

4. Скрипт *make_postproc.sh* получает на входе эти самые обработанные файлы с информацией и запускает питоновский скрипт *make_postproc.py*, который в свою очередь по обработанной информации строит все графики с помощью matplotlib`a

5. Скрипт *research.sh* принимает три аргумента шаг исследования STEP, максимальный размер массива MAXSIZE и количество замеров EXP и по очереди запускает все предыдущие скрипты, передавая им при необходимости аргументы STEP, MAXSIZE и EXP.

Результаты:

Результаты получены посредством выполнения команды:

./research.sh 500 10000 500

Далее в таблицах T — среднее время работы сортировки, RSE — относительная стандартная ошибка среднего

1. Адресная арифметика + уровень оптимизации O0

	Случайный массив			Упорядоченный массив		
Размер	T, мс	Замеров	RSE, %	T, мс	Замеров	RSE, %
1	0	500	0.40	0	500	0.66
1001	1.40	500	0.02	1.30	500	0.08
2001	5.44	500	0.04	5.14	500	0.01
3001	12.11	500	0.01	11.53	500	0.02
4001	21.39	500	0	20.46	500	0.01
5001	33.31	500	0.01	31.94	500	0.01
6001	47.82	500	0	45.96	500	0
7001	64.98	500	0	62.59	500	0

8001	84.76	500	0	81.90	500	0
9001	107.17	500	0	103.92	500	0.02
10001	132.31	500	0.01	128.52	500	0

2. Адресная арифметика + уровень оптимизации O2

	Случайный массив			Упорядоченный массив		
Размер	T, мс	Замеров	RSE, %	T, мс	Замеров	RSE, %
1	0	500	0.35	0	500	0.34
1001	0.37	500	0.19	0.30	500	0.01
2001	1.31	500	0.01	1.18	500	0.02
3001	2.85	500	0	2.64	500	0.01
4001	4.96	500	0.03	4.67	500	0.01
5001	7.66	500	0.03	7.29	500	0.03
6001	10.94	500	0.07	10.48	500	0.01
7001	14.78	500	0	14.26	500	0.01
8001	19.22	500	0.03	18.61	500	0.01
9001	24.23	500	0.01	23.54	500	0.01
10001	29.82	500	0.01	29.04	500	0

3. Индексация + уровень оптимизации O0

	Случайный массив			Упорядоченный массив		
Размер	T, мс	Замеров	RSE, %	T, мс	Замеров	RSE, %
1	0	500	0.28	0	500	0.40
1001	1.40	500	0.02	1.30	500	0.01
2001	5.44	500	0.01	5.14	500	0

3001	12.11	500	0.01	11.53	500	0.01
4001	21.39	500	0	20.45	500	0
5001	33.31	500	0	31.94	500	0.01
6001	47.82	500	0	45.97	500	0.01
7001	64.97	500	0	62.60	500	0
8001	84.74	500	0	81.89	500	0
9001	107.15	500	0	103.89	500	0
10001	132.28	500	0	128.51	500	0

4. Индексация + уровень оптимизации O2

	Случайный массив			Упорядоченный массив		
Размер	T, мс	Замеров	RSE, %	T, мс	Замеров	RSE, %
1	0	500	0.35	0	500	0.35
1001	0.36	500	0.10	0.30	500	0.01
2001	1.31	500	0.01	1.18	500	0.01
3001	2.85	500	0	2.64	500	0.07
4001	4.96	500	0.05	4.67	500	0.01
5001	7.66	500	0.06	7.29	500	0
6001	10.97	500	0.17	10.48	500	0
7001	14.79	500	0.03	14.25	500	0
8001	19.22	500	0.04	18.60	500	0
9001	24.23	500	0.02	23.54	500	0.01
10001	29.82	500	0	29.05	500	0.01

5. Указатели + уровень оптимизации O0

	Случайный массив			Упорядоченный массив		
Размер	T, мс	Замеров	RSE, %	T, мс	Замеров	RSE, %
1	0	500	0.28	0	500	0.84
1001	1.59	500	0	1.54	500	0
2001	6.24	500	0	6.13	500	0
3001	13.95	500	0	13.77	500	0
4001	24.71	500	0	24.46	500	0
5001	38.52	500	0	38.20	500	0
6001	55.38	500	0	55.00	500	0
7001	75.28	500	0	74.84	500	0
8001	98.26	500	0	97.74	500	0
9001	124.29	500	0	123.73	500	0
10001	153.46	500	0	152.79	500	0

6. Указатели + уровень оптимизации O2

	Случайный массив			Упорядоченный массив		
Размер	T, мс	Замеров	RSE, %	T, мс	Замеров	RSE, %
1	0	500	0.35	0	500	0.35
1001	0.65	500	0.01	0.59	500	0.04
2001	2.47	500	0.01	2.34	500	0.01
3001	5.47	500	0.05	5.24	500	0.01
4001	9.61	500	0.01	9.31	500	0.04
5001	14.92	500	0.01	14.52	500	0
6001	21.37	500	0	20.89	500	0

7001	28.99	500	0	28.43	500	0.01
8001	37.80	500	0.07	37.15	500	0.03
9001	47.71	500	0	46.99	500	0.02
10001	58.82	500	0	57.97	500	0

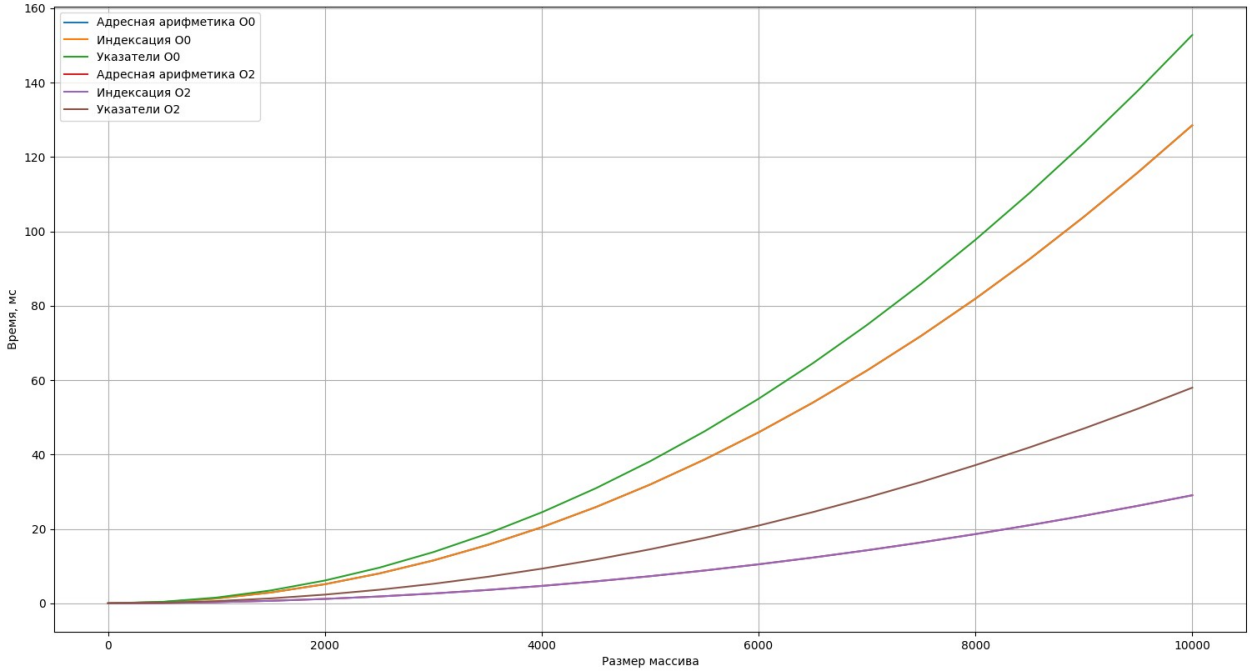

График зависимости времени выполнения программы от количества элементов массива для отсортированного массива. (График индексации совпадает с адресной арифметикой)

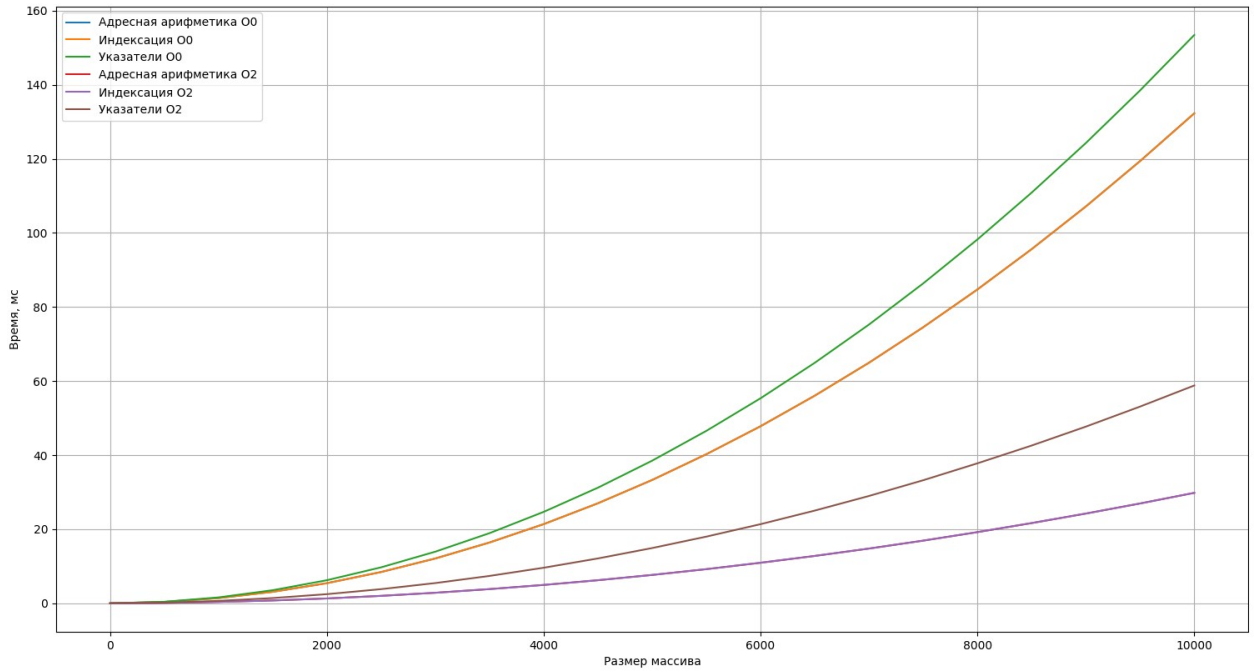

График зависимости времени выполнения программы от количества элементов массива для массива общего вида

Как можно заметить, время работы сортировки выбором при отсортированном массиве и при массиве общего вида примерно одинаковое. При сортировке выбором программа делает n^2 операций сравнения и n^2 swap`ов. То есть асимптотика и в лучшем и в худшем случае $O(n^2)$. Отсюда и такие близкие результаты на отсортированном массиве и на массиве общего вида.

Также сразу бросается в глаза, что реализация через указатели проигрывает во времени реализациям через адресную арифметику и через индексацию. Почему? Все дело в том, что «двигать» указатель более затратно по времени, чем увеличивать итерационную переменную. Т.е for (int *p = a; p < b; ++p) , где a и b указатели на первый элемент массива и «запоследний», работает дольше, чем for (int i = 0; i < n; ++i), где n размер массив

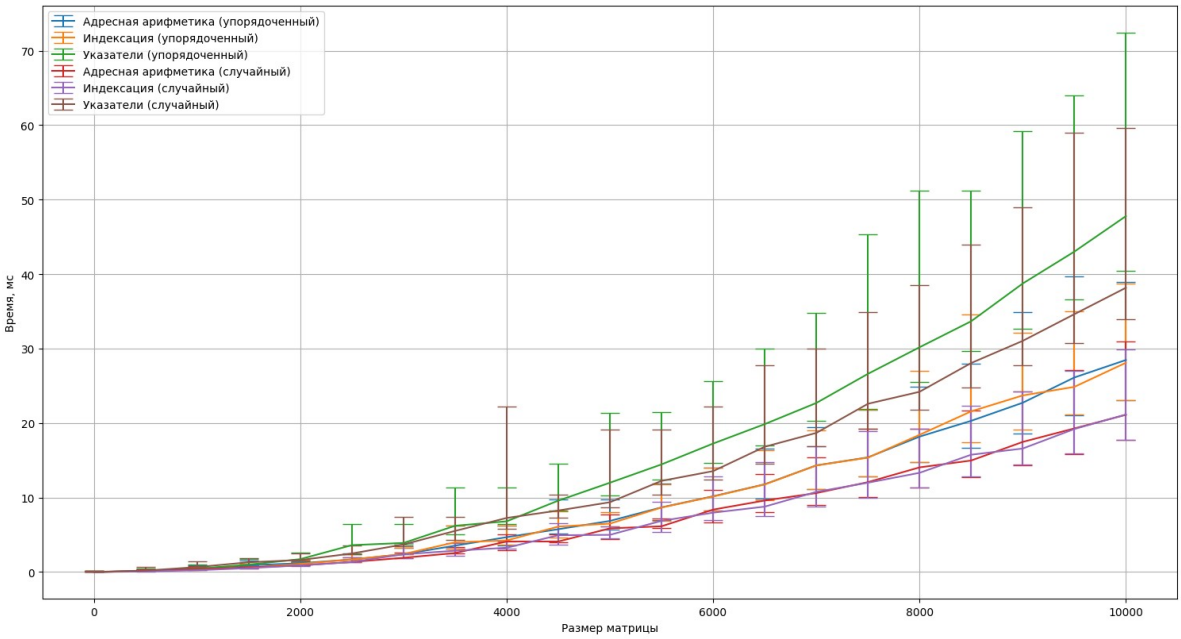

График с ошибкой для программ с уровнем оптимизации O2:

Как можно заметить по графику, иногда во время замеров случаются выбросы. Я так и не разобрался, с чем это связано. Возможно, с распределением памяти и доступом к ней.

Замеров	Размер	T, мс	RSE, %	Величина
500	1	0	0.35	0.83
500	1001	0.36	0.10	1.85
500	2001	1.31	0.01	1.92
500	3001	2.85	0	1.93
500	4001	4.96	0.05	1.95
500	5001	7.66	0.06	1.97
500	6001	10.97	0.17	1.94

500	7001	14.79	0.03	1.96
500	8001	19.22	0.04	1.97
500	9001	24.23	0.02	1.97
500	10001	29.82	0	-

Таблица для реализации с индексацией + оптимизация O2

Из таблицы видно, что «величина» растет с ростом размера массива. Это свидетельствует о квадратичности алгоритма сортировки.

Общий вывод:

В ходе исследования я выявил:

1. реализация сортировки выбором через указатели проигрывает по времени реализациям через индексацию и адресную арифметику.
2. сортировка выбором работает примерно одинаково на отсортированном массиве и на массиве общего вида.

Вопросы:

1. Вопрос: Какой способ быстрее и почему?

Ответ: Опираясь на данные своих исследований я могу сказать, что быстрее является реализация через индексацию.

2. Вопрос: В датасете обнаружена серия экспериментов с одним результатом. Можно ли заменить её одним экспериментом?

Ответ: Изучая датасет, я заметил, что серия экспериментов с примерно одним результатом присутствует при использовании O2 оптимизации. То есть вне зависимости от размера массива результаты замеров были одинаковыми. НО встречались и уникальные элементы, а соответственно заменить результаты одним замером нельзя.

3. Вопрос: Если заполнение случайными числами массива (или любая другая инициализация) присутствует в каждом эксперименте, то почему Вы замеряете время только у целевого алгоритма?

Ответ: Потому что если бы мы стали мерить время работы заполнения массива числами + время работы сортировки + время работы вывода — все бы это дало дополнительную погрешность, увеличилась бы вероятность выброса. Значит и результаты были бы менее точными, увеличилась бы RSE, пришлось бы проводить дополнительные замеры, что нерационально.