Анастасія Лазоренко

Практичне завдання №3

Завдання:

1. Експериментально визначити оптимальне значення змінної **min_size,** яка визначає максимальну довжину підмасиву для вигідного використання алгоритму Insertion Sort у модифікації Insertion-Quick sort.
2. Порівняти роботу алгоритмів сортування:
   Порівняти ефективність роботи наступних алгоритмів: Mergesort, Randomized Quicksort, Quicksort-Insertion sort, Three-Median Quicksort, Five-Median Quicksort.

Тестування алгоритмів проводитиметься на псевдо-випадково створених масивах (використовуючи метод randint() із вбудованого модуля random) для різної довжини масивів - від 2**10 до 2**20. Експеримент для кожної довжини повторюватиметься 10 разів, а для кінцевого аналізу чи перевірки доцільності певного значення min_size використовуватиметься середнє значення результату проміжних експериментів.
* *В другому завданні я також порівнювала ефективність стандартного алгоритму Quick Sort, тому на графіках є результати для шести алгоритмів.*

Технічні Специфікації:

Кількість ядер	4/8
Тактова частота	2,8 - 4,1 GHz
Оперативна пам’ять	16 Gb
OS	Ubuntu Xenial 16.04

Результати та висновки:

Task_1
Пошук оптимального значення довжини підмасиву для використання Insertion sort

Оскільки я вже мала результати для InsertionSort, я протестувала стандартний QuickSort на масивах довжиною 128 елементів. Порівнявши результати, впевнилась, що QuickSort працює швидше для вхідних даних такої довжини і тому подальший пошук оптимального значення для min_size проводила в межах (1, 128).
Для пошуку **більш точної верхньої межі** я використала принцип бінарного пошуку в функції b_search_max_lim(max_sure=128). Викликавши функцію 1000 разів, - знайшла абсолютні маскимальні значення для різних масивів та обчислила середнє максимальне значення **MAX_LIM ~ 52.**

Для оцінки переваги модифікації *quick_insertion_sort* над оригінальним *quicksort* я створювала словник **min_size_res** та записувала в нього результати експериментів, оновлюючи для кожної довжини масиву.
Ключами в словнику є значення min_size, а значеннями - відсоток виграного часу відносно часу виконання quick_sort. Тоді після експерименту я обираю максимальне

значення і зберігаю ключ у список, з якого потім отримую середнє значення оптимаьних min_size для певної довжини масиву.

Результат можна побачити у таблиці:

length of input array		min_size ~ *optimal size of sub array for Insertion sort*
	1024	8
	2048	11
	4096	8
	8192	6
	16384	7
	32768	6
	65536	6
	131072	7
	262144	7
	524288	7
	1048576	8

З таблиці видно, що значення оптимального min_size лежить в межах [6, 11] і містить повторювані значення. Узагальнене відображення залежності значення оптимального min_size від довжини вхідного масиву та лінію середнього спільного min_size можна побачити на графіку **optimal_min_size.png**
**(шкала осі x (розмір вхідного масиву) - логарифмічна → x = log_{2} length_of_array)*

Для підрахунку середнього значення для всіх тестованих масивів використано формулу зваженого середнього значення. Отже оптимальне обмеження довжини під-масиву для використання Insertion Sort ~ **min_size = 8**.

Отже, умову використання Insertion Sort можна записати так:

```
if (right - left + 1 <= 8) and (left < right)
```

*Також у файлі **min_size_search_results.txt** з детальними результатами виконання програми можн побачти, що у деяких випадках жодне значення min_size не було більш вигідним, ніж оригінальний Quick Sort. Тому навіть з визначеним оптимальним середнім значенням min_size алгоритм Insertion-Quick Sort іноді поступатиметься ефективністю простому Quick Sort (в залежності від комбінації елементів масиву та їх "зручності" для швидшого partition у Quick Sort)*

Task 2

Порівняти роботу алгоритмів сортування: Merge sort, Quick sort та його модифікацій

1. див. графік **2_time_graph.html**, таблицю **2_time_table.pdf**
З графіку залежності часу від довжини вхідного масиву для перелічених алгоритмів, алгоритм Insertion-Quick Sort виявився найшвидшим варіантом для роботи з масивами будь-якої тестованої довжини. Він також залишається найшвидшим, якщо додатково протестувати ці алгоритми на менших масивах (arr_len < 2**10).
*Хоча, він поступається оригінальному алгоритму Quick Sort на масивах довжиною від 2**10 до 2**14. Це можна можна пояснити тим, що масивам такої довжини відповідає більше оптимальне значення min_size у порівнянні з прийнятим для всіх оптимальним min_size = 8*
(див. графік optimal_min_size.png)

Для масивів довжиною <= 2**18 конкуренцію для Insertion-Quick Sort складає Three-Median Quick Sort, час роботи якого відрізняється лише на (0.001 -0.01)k. Проте, після 2**18 різниця починає збільшуватись майже вдвічі (на користь Insertion-Quick Sort).

З іншого боку, найбільший час виконнання належить алгоритму Merge Sort. Хоча його складність у найгіршому випадку краща, ніж у Quick Sort, алгоритм швидкого сортування та його модифікації працюють з різною складністю в залежності від підбору елементів масиву, тому в середньому все таки перевершує Merge Sort по часу виконання. *Цей факт також може бути пов'язаний з рекурсивною реалізацією Merge Sort, яка може потребувати додаткового часу для багатьох викликів функції.*

Five-Median Quick Sort працює швидше на великих масивах, і функція його часу виконання починає спадати на 2**18. Інші алгоритми в даному випадку поступаються наведеним оптимальним алгоритмам і в середньому працюють з невеликим відносним між собою відривом впродовж всього експерименту.

2. див. графік **2_comparisons_graph.html**, таблицю **2_comparisons_table.pdf**
На графіку залежності кількості порівнянь від розміру вхідного масиву видно, що більшість тестованих алгоритмів мають дуже близькі значення кількості порвнянь елементів. Варто відзначити, що найбільшу кількість порівнянь для усіх тестованих довжин масивів має Three-Median Quick Sort. Враховуючи таку особливість при використанні цього алгоритму варто мінімізувати складність самих порівнянь і надавати перевагу базовим типам даних для елементів масиву. На противагу Three-Median Quick Sort, найменшу кількість порівнянь здійснює Merge Sort, тож його оптимально буде використовувати для масивів зі складними типами елементів, які вимагають складніших операцій порівняння - таким чином менша кількість порівнянь певною мірою компенсує більшу затратність самої операції порівняння. *Також той факт, що Merge Sort стабільно має меншу кількість порівнянь підтверджує припущення, про рекурсію як причину довшого часу виконання.*