Прикладні алгоритми. Завдання 1, звіт

Михайло Голуб

11 вересня 2024 р.

Реалізація класу множина з сортуванням:

Для реалізації елементу множини на мові програмування Python створено клас *Node*, що містить значення *value* та *next_node*. Перше містить значення елементу множини, інше вказівник на наступний *Node* (або *None* якщо наступного *Node* немає).

Для реалізації множини створено клас *Set*, що містить значення *first_node* та методи для операцій над собою: *insert, __str__, delete, search, clear, __len__, to_list*; та іншим представником класу: *union, intersection, set_difference, sym_difference*.

Реалізація методів класу *Set*:

insert – цей метод додає значення до множини за наступним алгоритмом роботи:

1. Якщо не існує першого елемента – створити його та записати туди значення, завершити;
2. Якщо значення нового елемента менше значення першого елемента – створити новий елемент,зробити новий елемент першим, перший – другим, завершити;
3. Якщо значення нового елемента рівне першому – завершити;
4. Крокувати через усі елементи множини: якщо обраний елемент не існує – вставити новий елемент, завершити; якщо обраний елемент менше обраного – крокувати далі; якщо обраний елемент більше обраного – вставити новий елемент перед ним, завершити; якщо обраний та новий елемент мають однакове значення – завершити.

Кроки 1-3 є частковим випадком 4, оскільки крокування можливо почати лише з 2го елементу не ускладнюючи код циклу.

__str__ – цей метод крокує через усі елементи множини, записує їх у *list*, потім переганяє їх у *str* та повертає отраману строку.

to list – цей метод крокує через усі елементи множини, записує їх у *list* та повертає його.

delete – цей метод видаляє значення з множини за наступним алгоритмом роботи:

1. Якщо множина пуста – завершити;
2. Якщо значення менше першого елемента – завершити;
3. Якщо значення рівне першому елементу – видалити його, зробити другий елемент першим, завершити;

4. Крокувати через усі елементи множини: якщо обраний елемент не існує – завершити; якщо значення рівне значенню обраного елемента – видалити його, зв'язати сусідні елементи між собою, завершити; інакше – крокувати далі.

search – цей метод перевіряє наявність елемента в множині за наступним алгоритмом роботи:

1. Якщо множина пуста – повернути -1;
2. Якщо значення менше першого елемента – повернути -1;
3. Якщо значення рівне першому елементу – повернути 0;
4. Крокувати через усі елементи множини: якщо обраний елемент не існує – повернути -1; якщо значення рівне значенню обраного елемента – повернути його позицію; інакше – крокувати далі.

clear – цей метод видаляє відв'язує перший елемент від множини, після чого його десь там підбирає garbage collector.

Бінарні методи класу множини реалізовані за загальним принципом:

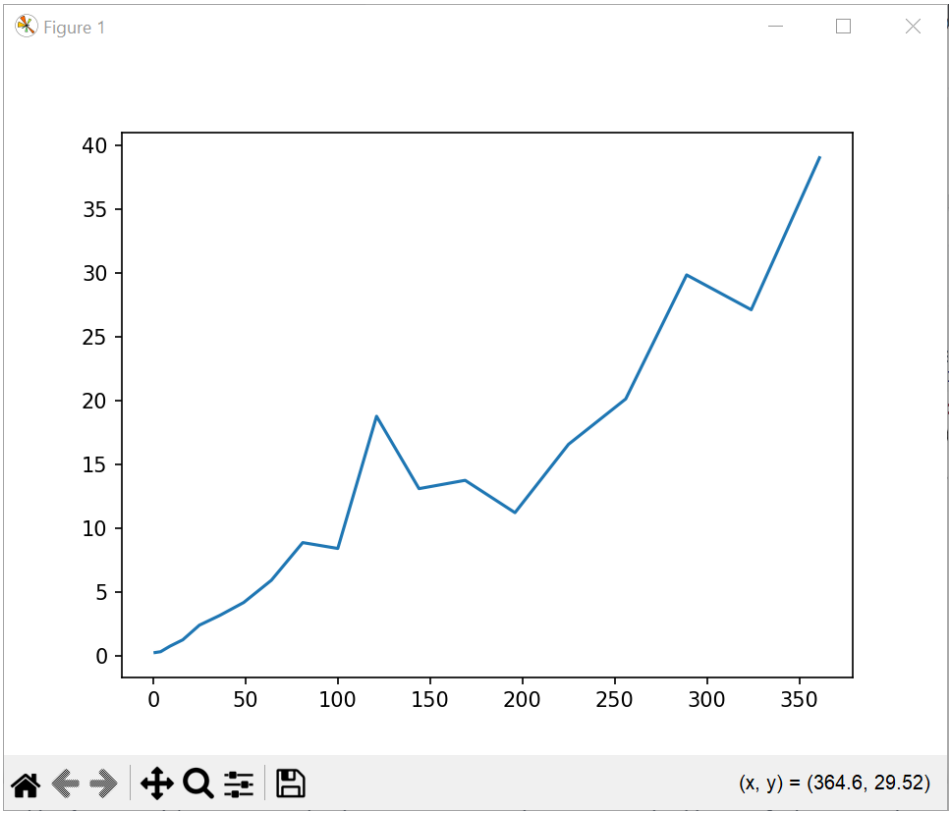
1. Виконати відповідну дію якщо одна або обидві з множин пусті
2. Крокувати через обидві множини: порівнювати обрані елементи, якщо виконується умова операції – виконати дію; інакше – зробити крок в множині чий обраний елемент менший

Детальний опис цих методів займе більше часу ніж усе інше сумарно, тому він буде пропущений.

Тестування швидкості роботи:

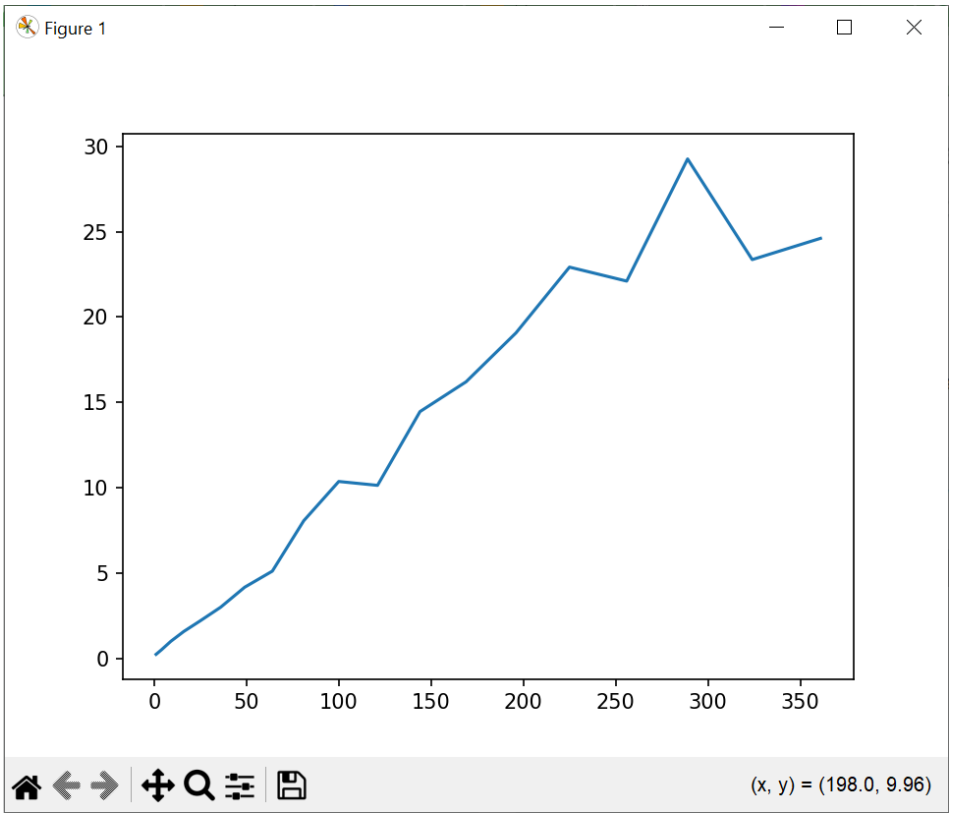
Дядько Фестер (Аддамс), представник класу *SkilledTester*, проводить тестування наступним чином: отримує два "синхронізовані"масиви: "потужність множин"та "кількість тестувань"; тестує, та повертає масив середніх часів виконання. Після цього створюється графік залежності часу виконання (в мкс) від потужності множини.

Результати тестування *search* елементів яких немає в множині:



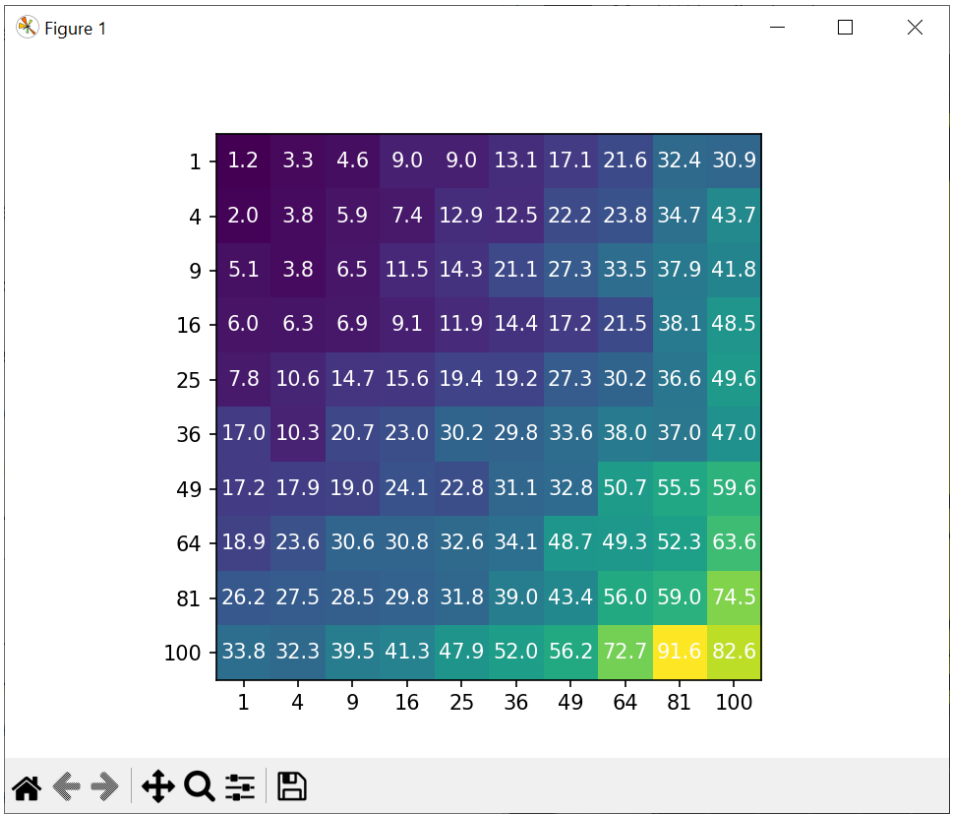
Протестовано 19 вибірок потужністю від 1 до 361. Не дивлячись на малу кількість точок та значне коливання значень, видно що залежність часу виконання від потужності схожа на лінійну функцію. Це очікувано, оскільку алгоритм не має вкладених циклів і в середньому має складність $O(\frac{n}{2})$

Результати тестування *search* елементів які є в множині:



Протестовано 19 вибірок потужністю від 1 до 361. Не дивлячись на малу кількість точок та значне коливання значень, видно що залежність часу виконання від потужності схожа на лінійну функцію. Це очікувано, оскільку алгоритм не має вкладених циклів і в середньому має складність $O(\frac{n}{2})$

Результати тестування *union* множин:



Протестовано 100 пар вибірок множин потужністю від 1 до 100. З heatmap видно, що збільшення сумарної кількості елементів множин в 2 рази призводить збільшення часу в 1.7-2.5 рази. Це очікувано, оскільки алгоритм не може зробити ітерацій більше ніж є сумарно елементів у множинах, а отже має найгіршу складність $O(n_1 + n_2)$

Посилання на репозиторій практикуму:
https : *//github.com/MINIAProgramStudio/applied_algorythms/tree/main/task_1*