**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ**

**УКРАЇНИ «КПІ ім. Ігоря Сікорського»**

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування і спеціалізовних комп’ютерних систем**

**КУРСОВА РОБОТА**

***з дисципліни «Структури даних та алгоритми»***

Виконав: Брюханов О.С.

Група: КВ-14

Номер залікової книжки: ???

Допущений до захисту

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**2 семестр 2021/2022 навч. року**

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ**

**УКРАЇНИ «КПІ ім. Ігоря Сікорського»**

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування і спеціалізовних комп’ютерних систем**

Узгоджено Захищена "\_\_" \_\_\_\_\_\_ 20\_\_р.

Керівник роботи з оцінкою \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_ /О.І. Марченко/ \_\_\_\_\_\_\_\_\_\_\_\_ /О.І. Марченко/

***Дослідженя ефективності методів сортування (сортування методом вибору №6, №8 та сортування обміном та вибором (гібрид) №3) на багатовимірних масивах***

Виконавець роботи: \_\_\_\_\_\_\_\_\_\_

(підпис)

Брюханов Олександр Сергійович

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_р.

Технічне завдання на курсову роботу

**I.** Описати теоретичні положення, від яких відштовхується дослідження, тобто принцип та схему роботи кожного із досліджуваних алгоритмів сортування для одновимірного масива, навести загальновідомі властивості цих алгоритмів та оцінки кількості операцій порівняння та присвоєння для них.

**II.** Скласти алгоритми рішення задачі сортування в багато-вимірному масиві заданими за варіантом методами та написати на мові програмування за цими алгоритмами програму, яка відповідає вимогам розділу «Вимоги до програми курсової роботи».

**III.** Виконати налагодження та тестування коректності роботи написаної програми.

**IV.** Провести практичні дослідження швидкодії складених алгоритмів, тобто виміри часу роботи цих алгоритмів для різних випадків та геометричних розмірів багатовимірних масивів.

**V.** За результатами досліджень скласти порівняльні таблиці за різними ознаками.

Одна таблиця результатів (вимірів часу сортування впорядкованого, випадкового і обернено-впорядкованого масива) для масива з заданими геометричними розмірами повинна бути такою:

Таблиця № для масива A[P,M,N], де P= ; M= ; N= ;

|  |  |  |  |
| --- | --- | --- | --- |
|  | Впорядкований | Невпорядкований | Обернено впорядкований |
| Назва алгоритму 1 |  |  |  |
| Назва алгоритму 2 |  |  |  |
| Назва алгоритму 3 |  |  |  |

Для варіантів курсової роботи, де крім алгоритмів порівнюються також способи обходу, в назвах рядків таблиць потрібно вказати як назви алгоритмів, так і номери способів обходу.

Для виконання ґрунтовного аналізу алгоритмів потрібно зробити виміри часу та побудувати таблиці для декількох масивів з різними геометричними розмірами.

Зробити виміри часу для стандартного випадку одномірного масива, довжина якого вибирається такою, щоб можна було виконати коректний порівняльний аналіз з рішенням цієї ж задачі для багатовимірного масива.

Кількість необхідних таблиць для масивів з різними геометричними розмірами залежить від задачі конкретного варіанту курсової роботи і вибираються так, щоб виконати всебічний та ґрунтовний порівняльний аналіз заданих алгоритмів.

Рекомендації випадків дослідження з різними геометричними розмірами масивів наведені у розділі «Випадки дослідження».

**VI.** Для наочності подання інформації за отриманими результатами рекомендується також будувати стовпчикові діаграми та графіки.

**VII.** Виконати порівняльний аналіз поведінки заданих алгоритмів за отриманими результатами (вимірами часу):

* для одномірного масива відносно загальновідомої теорії;
* для багатовимірних масивів відносно результатів для одномірного масива;
* для заданих алгоритмів на багатовимірних масивах між собою;
* дослідити вплив різних геометричних розмірів багатовимірних масивів на поведінку алгоритмів та їх взаємовідношення між собою;
* **для всіх вищезазначених пунктів порівняльного аналізу пояснити, ЧОМУ алгоритми в розглянутих ситуаціях поводять себе саме так, а не інакше**.

**VIII.** Зробити висновки за зробленим порівняльним аналізом.

**IX.** Програму курсової роботи під час її захисту ОБОВ’ЯЗКОВО мати при собі на електронному носії інформації.

Варіант №124

Задача

Впорядкувати окремо кожен переріз тривимірного масива Arr3D[P, M, N] таким чином: переставити стовпчики перерізу за незменшенням сум їх елементів.

Досліджувані методи та алгоритми

1. Алгоритм сортування №6 методом прямого вибору
2. Алгоритм сортування №8 методом прямого вибору
3. Гібридний алгоритм «Вибір№3-обмін»

Способи обходу

1. В якості першого етапу сортування сформувати додатковий вектор Sum, довжина якого дорівнює кількості стовпчиків і значеннями якого є суми елементів відповідних стовпчиків. Використовуючи елементи вектора Sum як ключі сортування, переставляти відповідні стовпчики кожен раз, коли треба переставляти ключі. При перестановці стовпчиків потрібно саме копіювати їх елементи, а не копіювати вказівники на них, використовуючи операції з вказівниками мови C/C++.

Теоретичні положення

Загальна характеристика алгоритмів сортування прямого вибору

Алгоритми сортування вибором ділять масив на відсортовану та невідсортовану частину. На почтаку сортування весь масив вважається за одну велику невідсортовану частину, яка потім розбивається на відсортовану та невідсортовану. В процессі сортування невідсортована частина зникає.

**Принцип роботи:**

1. У діапазоні індексів від i до n - 1 шукаємо найменьший елемент.
2. Міняємо його місяцями із iтим елементом масиву. Таким чином утворюється відсортована частина.

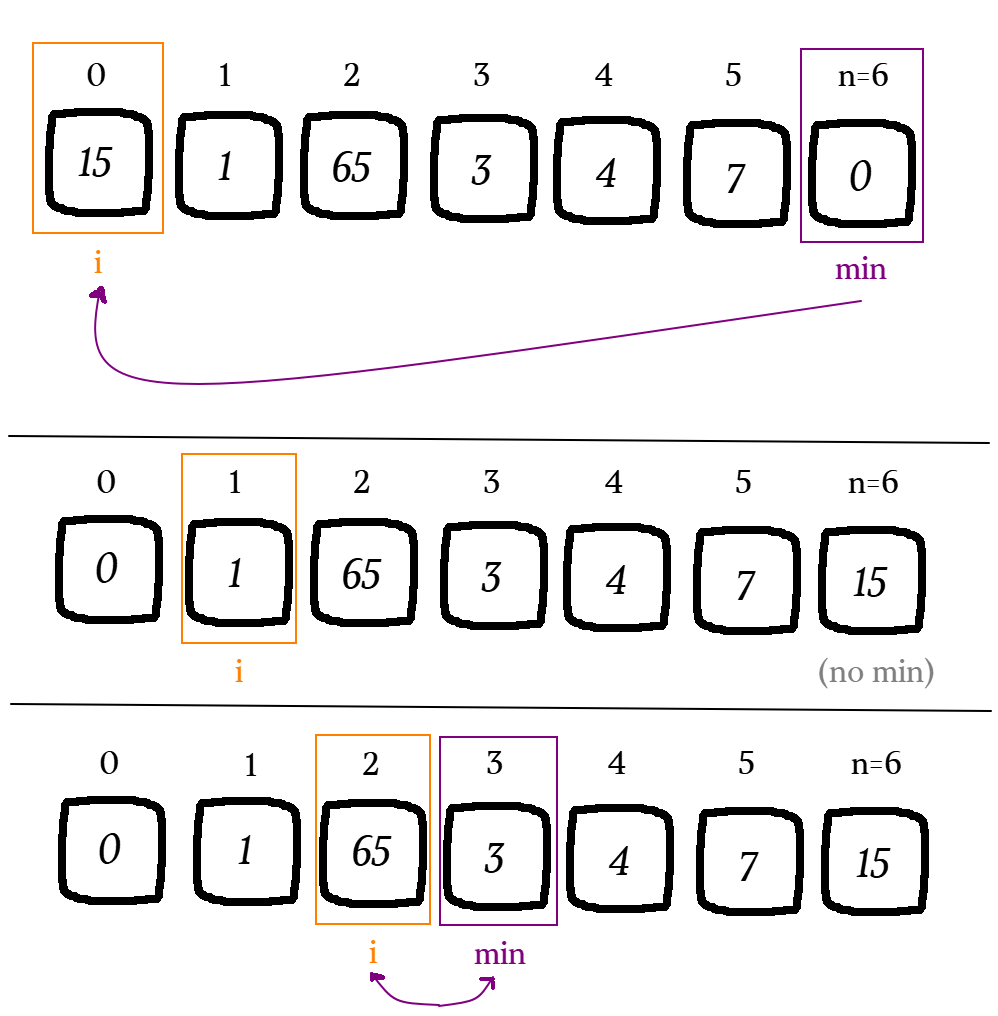
Ці пункти повторюються для кожного натурального i від 0 до n - 2. Таким чином відсортована частина постійно зростає до тих пір, поки повністю не поглине невідсортовану.

Алгоритм сортування методом прямого вибору №6

Цей алгоритм не сильно відрізняється від загального алгоритму сортування вибором, принцип роботи якого описано вище. Алгоритм №6 утворює відсортовану частину з лівої сторони масиву та нарощує її за допомогою перміщення до неї найменшого елементу з невідсортованої частини.

**Принцип роботи:**

1. Створюємо індекс i = 0, min = i.
2. Шукаємо елемент у диапазоні індексів від i + 1 до N найменший елемент, який також менше елементу з індексом min.
3. Якщо такий елемент знайдений (тобто min != i), то міняємо місцями елементи з індексами i та min.
4. Інкрементуємо i.
5. Повторюємо пункти 1-4 доки i не дійде до N – 2 включно.



**Алгоритм на С:**

clock\_t Select6(int \* A, int N) {

int imin, tmp;

clock\_t time\_start, time\_stop;

time\_start = clock();

for (int s = 0; s < N - 1; s++) {

imin = s;

for (int i = s + 1; i < N; i++)

if (A[i] < A[imin]) imin = i;

if (imin != s) {

tmp = A[imin];

A[imin] = A[s];

A[s] = tmp;

}

}

time\_stop = clock();

return time\_stop - time\_start;

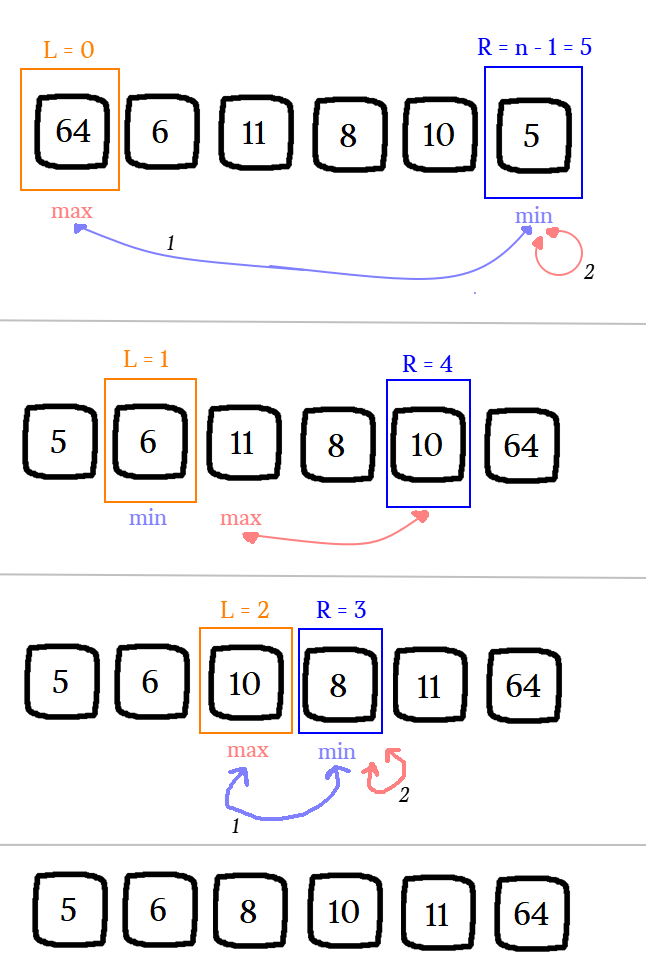
}

Алгоритм сортування методом прямого вибору №8

За загальною характеристикою таких алгоритмів, нам відомо, що у масиві буде невідсортована та відсортована частина. Особливість восьмого алгоритму полягає у тому, що відсортованих частин не одна, а дві: зліва та зправа від невідсортованої. Найменший та найбільший елемент невідсортованої частини йде у ліву та праву відсортовану частину відповідно. У процессі сортування невідсортована частина у центрі масиву поступово зменшується, поки її не поглинуть дві відсортовані.

**Принцип роботи:**

1. Визначаємо індекси L = 0 та R = n – 1.
2. Приймаємо за найменший та найбільший елемент той, який з індексом L.
3. Шукаємо у діапазоні ключів від L + 1 до R включно найбільший та найменший елемент. Шукані елементи, звісно, мають бути відповідно більше або менше елемента з індексом L.
4. Якщо є мінімальний елемент, то міняємо його місцями з індексом L.
5. Якщо є максимальний елемент, то міняємо його місцями з R.
6. L інкрементуємо, а R декрементуємо.
7. Повторюємо пункти 1-6 доки L не перестане бути менше R (поки є невідсортована частина).



**Алгоритм на С:**

clock\_t Select8(int \* A, int N) {

int L, R, imin, imax, tmp;

clock\_t time\_start, time\_stop;

time\_start = clock();

L = 0;

R = N - 1;

while (L < R) {

imin = L;

imax = L;

for (int i = L + 1; i < R + 1; i++)

if (A[i] < A[imin]) imin = i;

else

if (A[i] > A[imax]) imax = i;

if (imin != L) {

tmp = A[imin];

A[imin] = A[L];

A[L] = tmp;

}

if (imax != R) {

if (imax == L) {

tmp = A[imin];

A[imin] = A[R];

A[R] = tmp;

} else {

tmp = A[imax];

A[imax] = A[R];

A[R] = tmp;

}

L = L + 1;

R = R - 1;

}

time\_stop = clock();

return time\_stop - time\_start;

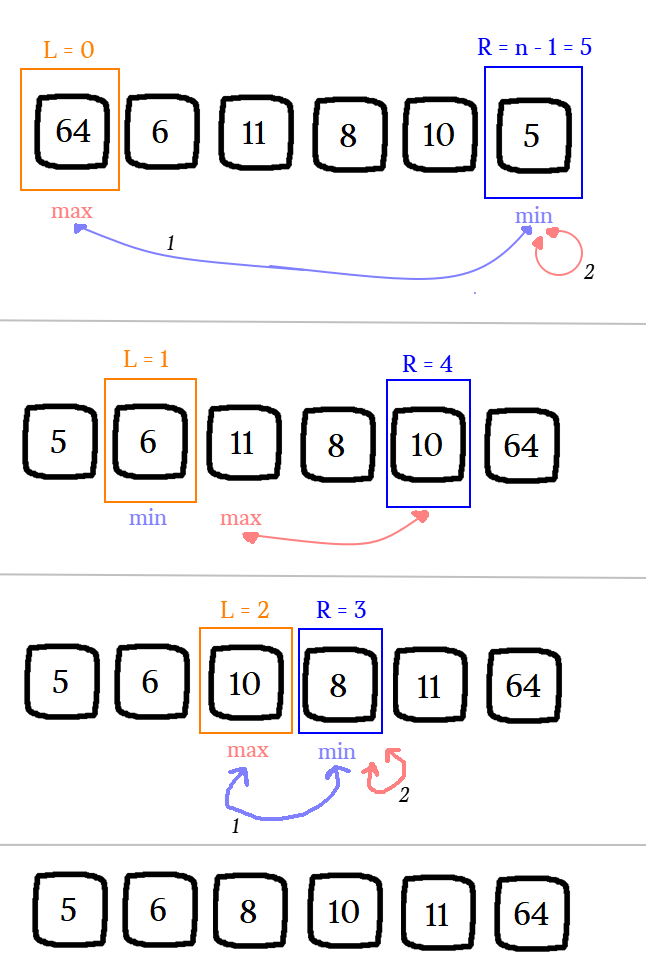
}

Гібридний алгоритм сортування вибором/обміном №3

Це дуже дивний алгоритм, гібрид класичного сортування обміном та сортування вибром №3. Цей алгоритм нагадує восьмий тим, що у ньому дві відсортовані частини оточують невідсортовану усередені. Але після кожного звужування (або початку виконання) елементи по краям невідсортованої частини міняються місцями, якщо перший більше останнього. В решті його поведінка подібна до попереднього алгоритму, за винятком того, що замість запам’ятовування індексу найменьшого/найбільшого елементу, цей алгоритм зберігає значення цих елементів та виконує перестановку min <-> L, max <-> R кожен раз, а не у кінці внутрішнього циклу, що зменшує кількість висмоктувань даних, але значано збільшує кількість перстановок.

**Принцип роботи:**

1. Визначити індекси L = 0, R = n - 1.
2. Визначити змінні min = arr[L], max = arr[R].
3. Якщо, елемент з індексом L більше елемента з індексом R, поміняти їх місцями.
4. Для кожного елементу в діапазоні індексів від L + 1 до R включно, виконати наступне:
5. Якщо елемент менше min, то поміняти його місцями з L та зберегти його значення у min.
6. Або, якщо елемент більше max, то поміняти його місцями з R та зберегти його значення у max.
7. L інкрементуємо, а R декрементуємо.
8. Повторюємо пункти 1-5 доки L не перестане бути менше R (поки є невідсортована частина).



**Алгоритм на С:**

clock\_t Select3Exchange(int \* A, int N) {

int Min, Max, tmp;

int L, R;

clock\_t time\_start, time\_stop;

time\_start = clock();

L = 0;

R = N - 1;

while (L < R) {

if (A[L] > A[R]) {

tmp = A[L];

A[L] = A[R];

A[R] = tmp;

}

Min = A[L];

Max = A[R];

for (int i = L + 1; i < R + 1; i++) {

if (A[i] < Min) {

Min = A[i];

A[i] = A[L];

A[L] = Min;

} else if (A[i] > Max) {

Max = A[i];

A[i] = A[R];

A[R] = Max;

}

}

L = L + 1;

R = R - 1;

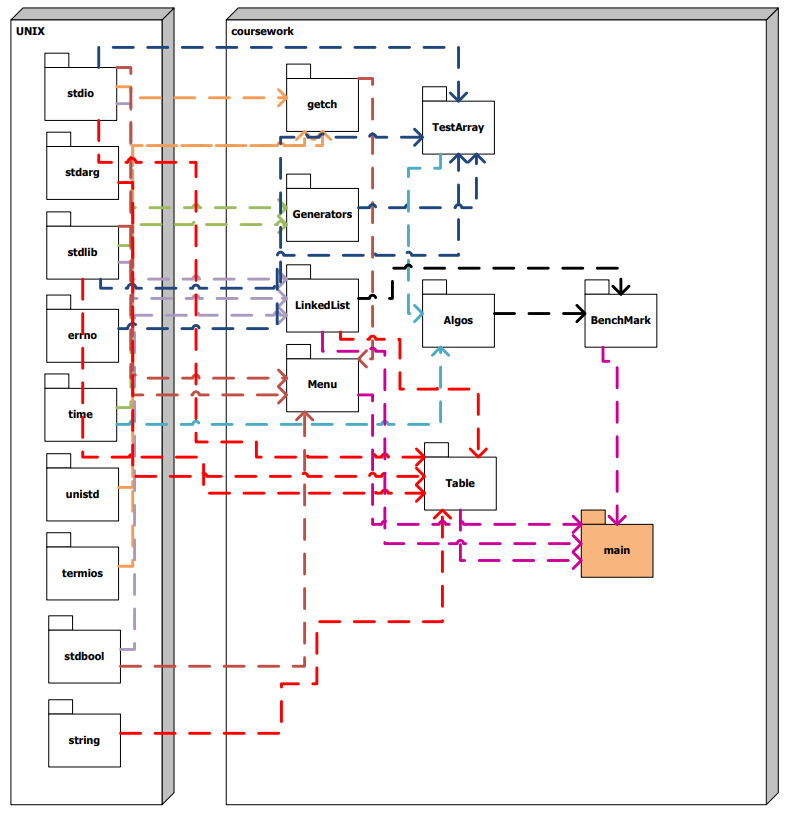
}

time\_stop = clock();

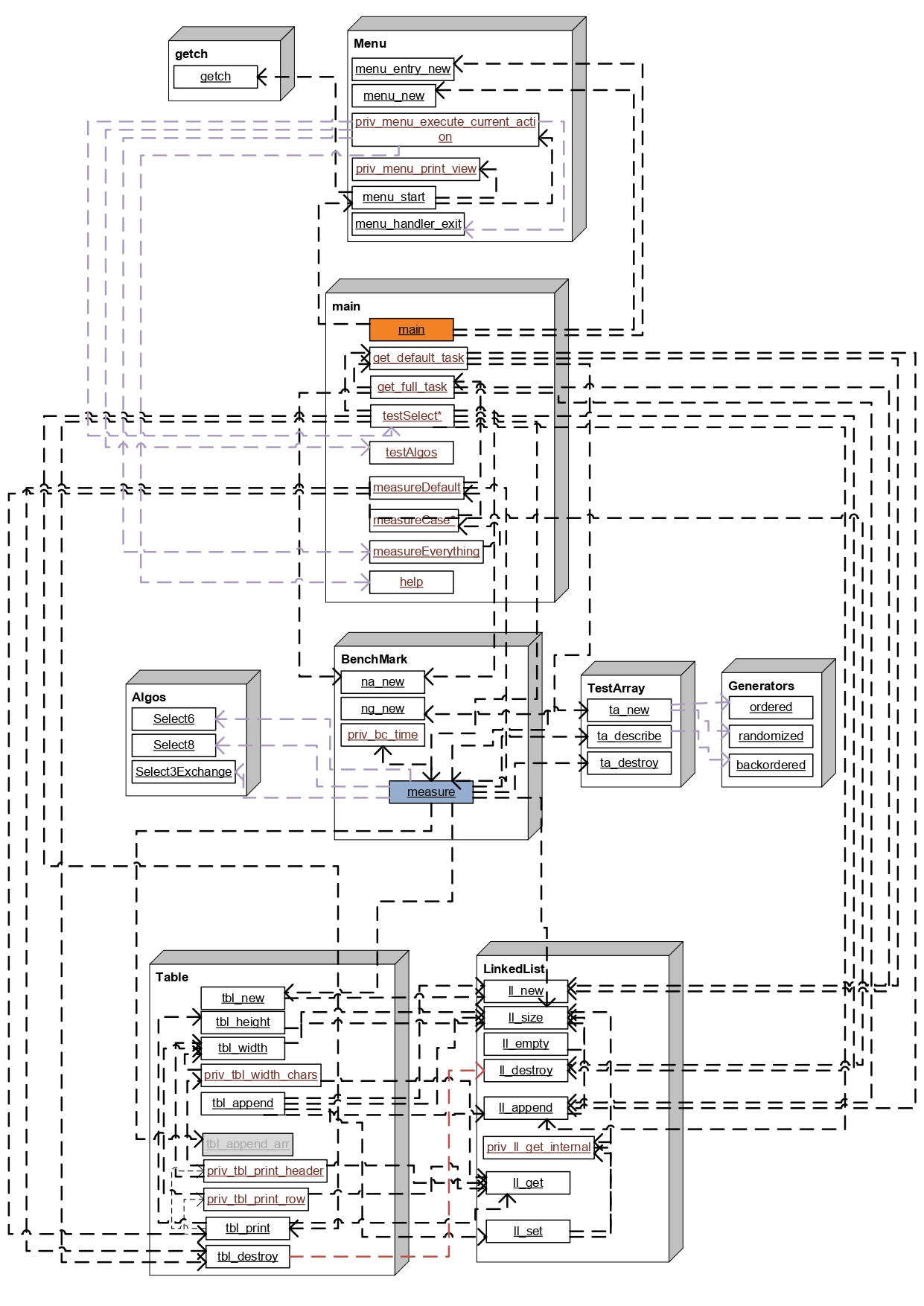
return time\_stop - time\_start;

}

Схема імпорту/експорту модулів



Структурна схема взаємовикликів функцій



Опис та призначення процедур та функцій

|  |
| --- |
| coursework |

|  |
| --- |
| Menu (Menu.c, Menu.h) |
| menu\_entry\_new |
| Створює структуру MenuEntry за параметрами та повертає вказівник на неї. Дозволяє не створювати структуру мануально, що є важливим, адже створення опцій меню відбувається у програмі часто. |
| menu\_ new |
| Створює структуру Menu, яка необхідна для роботи функцій, що працюють з меню. |
| menu\_handler\_exit |
| Функція, яка завжди повертає false. Меню закривається тоді, коли обробник повертає 0, ця функція дозволяє легко створювати опцію виходу з меню. |
| priv\_menu\_execute\_current\_action |
| Виконує обробник опції меню, яка була обрана користувачем. Ця функція призначена для внутрішнього використання та змінює змінну структури willExit. |
| priv\_menu\_print\_view |
| Відчищує екран та виводить меню. Ця функція призначена для внутрішнього використання. |
| menu\_start |
| Запускає меню. Повертає завжди 0, але тільки тоді, коли щось змусить меню закритися (willExit стане true => обробник поверне 0). |

|  |
| --- |
| Table (Table.c, Table.h) |
| tbl\_new |
| Створює структуру Table та повертає вказівник на неї. |
| tbl\_height |
| Повертає кількість рядків у таблиці. |
| tbl\_width |
| Повертає кількість стовпців у таблиці. |
| priv\_tbl\_width\_chars |
| Повертає ширину таблиці у знаках. Ця функція призначена для внутрішнього використання. |
| tbl\_append |
| Додає рядок у таблицю. |
| tbl\_append\_arr |
| Додає рядок у таблицю, але з масиву, а не аргументів. |
| priv\_tbl\_print\_header |
| Виводить заголовок таблиці на екран. Ця функція призначена для внутрішнього використання. |
| priv\_tbl\_print\_row |
| Виводить рядок таблиці на екран. Ця функція призначена для внутрішнього використання. |
| tbl\_print |
| Виводить усю таблицю на екран. |
| tbl\_destroy |
| Звільнює пам’ять зайняту таблицею, списками та даними в таблиці. |

|  |
| --- |
| LinkedList (LinkedList.c, LinkedList.h) |
| ll\_new |
| Створює структуру двозв’язного списку та повертає вказівник на неї. |
| ll\_size |
| Повертає к-сть елементів у списку. |
| ll\_empty |
| Повертає 1, якщо у списку немає елементів, 0 якщо у списку є елементи. |
| ll\_append |
| Додає у список елемент, який зберігає у собі вказівник на дані. |
| ll\_destroy |
| Звільнює пам’ять виділену під список, його елементи **та дані, на які вказують ці елементи.** |
| priv\_ll\_get\_internal |
| Повертає елемент списку за індексом (може бути від’ємним) або NULL, якщо такого елементу немає у списку (при цьому в STDERR з’явиться повідомлення про помилку). Ця функція призначена для внутрішнього використання. |
| ll\_get |
| Повертає вказівник на дані елементу списку за індексом. |
| ll\_set |
| Встановлює вказівник на дані елементу списка за індексом. |

|  |
| --- |
| TestArray (TestArray.c, TestArray.h) |
| ta\_new |
| Створює 3-вимірний масив із розмірами PxMxN, наповнює його генеруючою функцією seed та повертає вказівник на перший переріз масиву. |
| ta\_describe |
| Виводить перерізи масиву на екран. Корисно для зневаджування, використовується у режимі перевірки роботи алгоритмів. |
| ta\_destroy |
| Звільнює пам’ять, виділену під масив (з його перерізами, стовпцями та рядками). |

|  |
| --- |
| Generators (Generators.c, Generators.h) |
| Ці файли визначають генеруючи функції, якими заповнюється масив. Генеруючі функції отримують як аргумент попередне значення (0, якщо такого нема) масиву та повертають наступне. |
| ordered |
| Поветрає значення за неспаданням (кожне значення на 1 більше попереднього на множині значень беззнакового цілого). |
| backordered |
| Перше значення дорівнює P\*M\*N, кожне наступне менше попереднього на 1. |
| randomized |
| Повертає значення, отримане за допомогою системного виклику rand. Максимальне значення дорівнює P\*M\*N, але його повернення не гарантується. |

|  |
| --- |
| BenchMark (BenchMark.c, BenchMark.h) |
| na\_new |
| Повертає вказівник на структуру NamedAlgo, яка зберігає рядок з іменем алгоритму сортування та вказівник на алгоритм. Для алгоритмів, в яких ім’я не відрізняється від назви їх функції, є макрос ALGO(х), що розкривається як na\_new(“x”, &x). |
| ng\_new |
| Повертає вказівник на структуру NamedGenerator, яка зберігає рядок з іменем генеруючої функції та вказівник на неї. Для функцій, в яких ім’я не відрізняється від назви, є макрос SEED(х), що розкривається як ng\_new(“x”, &x). |
| priv\_bc\_time |
| За методикою, викладеною на лекціях та в методичці, повертає середне значення вимірів швидкодії алгоритмів. Для внутрішнього використання в модулі, не експортується. |
| measure |
| Виконує завдання виміру та повертає таблицю з результатами. Оскільки параметрів завдання виміру дуже багато, ця функція приймає як єдиний аргумент структуру BenchMarkTask, яка описує що саме і яким чином потрібно виміряти:   |  |  | | --- | --- | | BenchMarkTask | | | *LinkedList\*<NamedAlgo\*>* **algos** | Двозв’язний список із структур NamedAlgo, які вказують які саме алгоритми потрібно протестувати на швидкодію. | | *LinkedList\*<NamedGenerator\*>* **seeders** | Двозв’язний список із структур NamedGemerator, які вказують які саме наповнювачі алгоритму потрібно використати. | | *unsigned int* **multiplier** | Число на яке потрібно помножити результати вимірювань. | | *unsigned int* **iterations** | Кількість вимірювань, які потрібно зробити. Має бути як мінімум 12, але рекомендовано 28. | | *unsigned int* **size***[3]* | Масив із розміром 3д масиву {P, M, N} | | *bool* **describe** | Якщо встановлено у 1, то після сортування буде викликано ta\_describe (на останній ітерації). | |

|  |
| --- |
| getch (getch.c, getch.h) |
| getch |
| Імплементація функції getch з conio.h для UNIX. |

|  |
| --- |
| main.c |
| get\_default\_task |
| Повертає вказівник на завдання виміру без алгоритмів, з усіма генеруючими функціями, множником 1.0 та к-стю ітерацій 12. |
| get\_full\_task |
| Повертає вказівник на завдання виміру з усіма алгоритмами, генеруючими функціями, множником 1.0 та к-стю ітерацій 28. |
| testSelect6 |
| В інтерактивному режимі запитує розмір масиву в користувача та виводить результати вимірів алгоритму сортування вибором №6. |
| testSelect8 |
| В інтерактивному режимі запитує розмір масиву в користувача та виводить результати вимірів алгоритму сортування вибором №8. |
| testSelect3Exchange |
| В інтерактивному режимі запитує розмір масиву в користувача та виводить результати вимірів алгоритму сортування вибором+обміном №3. |
| testAlgos |
| Виводить меню тестування коректності роботи алгоритмів. |
| measureDefault |
| Виконує виміри повним заваданням виміру (get\_full\_task) та розміром size, який має бути заздалегідь визначеним. Також виводить таблицю з результатами. |
| measureCase1 |
| Виконує виміри для першого випадку дослідження та виводить результати. |
| measureCase2 |
| Виконує виміри для другого випадку дослідження та виводить результати. |
| measureCase3 |
| Виконує виміри для третього випадку дослідження та виводить результати. |
| measureEverything |
| Послідовно викликає measureCase1, measureCase2, measureCase3. |
| help |
| Виводить інструкції по користуваня програмою. |
| main |
| Ініціалізує генератор випадкових чисел та виводить головне меню програми. |

Текст програми

main.c

#include "LinkedList.h"

#include "BenchMark.h"

#include "Table.h"

#include "Menu.h"

unsigned int\* size;

\_\_attribute\_\_((flatten)) struct BenchMarkTask\* get\_default\_task() {

struct LinkedList\* seeds = ll\_new();

ll\_append(seeds, SEED(ordered));

ll\_append(seeds, SEED(randomized));

ll\_append(seeds, SEED(backordered));

struct BenchMarkTask\* task = malloc(sizeof(struct BenchMarkTask));

task->seeders = seeds;

task->describe = false;

task->multiplier = 1;

task->iterations = 12;

return task;

}

\_\_attribute\_\_((flatten)) struct BenchMarkTask\* get\_full\_task() {

struct LinkedList\* algos = ll\_new();

ll\_append(algos, ALGO(Select6));

ll\_append(algos, ALGO(Select8));

ll\_append(algos, ALGO(Select3Exchange));

struct BenchMarkTask\* task = get\_default\_task();

task->algos = algos;

task->iterations = 28;

return task;

}

MENU\_HANDLER(testSelect6) {

struct LinkedList\* algos = ll\_new();

ll\_append(algos, ALGO(Select6));

SETSIZE(size, 1, 0, 0);

printf("P = const = 1\nM = ");

fflush(stdout);

scanf("%d", &size[1]);

printf("N = ");

fflush(stdout);

scanf("%d", &size[2]);

struct BenchMarkTask\* task = get\_default\_task();

task->algos = algos;

task->describe = true;

task->size = size;

struct Table\* results = measure(task);

tbl\_print(results);

tbl\_destroy(results);

ll\_destroy(algos);

ll\_destroy(task->seeders);

free(task);

printf("\a\n");

system("pause");

return true;

}

MENU\_HANDLER(testSelect8) {

struct LinkedList\* algos = ll\_new();

ll\_append(algos, ALGO(Select8));

SETSIZE(size, 1, 0, 0);

printf("P = const = 1\nM = ");

fflush(stdout);

scanf("%d", &size[1]);

printf("N = ");

fflush(stdout);

scanf("%d", &size[2]);

struct BenchMarkTask\* task = get\_default\_task();

task->algos = algos;

task->describe = true;

task->size = size;

struct Table\* results = measure(task);

tbl\_print(results);

tbl\_destroy(results);

ll\_destroy(algos);

ll\_destroy(task->seeders);

free(task);

printf("\a\n");

system("pause");

return true;

}

MENU\_HANDLER(testSelect3Exchange) {

struct LinkedList\* algos = ll\_new();

ll\_append(algos, ALGO(Select3Exchange));

SETSIZE(size, 1, 0, 0);

printf("P = const = 1\nM = ");

fflush(stdout);

scanf("%d", &size[1]);

printf("N = ");

fflush(stdout);

scanf("%d", &size[2]);

struct BenchMarkTask\* task = get\_default\_task();

task->algos = algos;

task->describe = true;

task->size = size;

struct Table\* results = measure(task);

tbl\_print(results);

tbl\_destroy(results);

ll\_destroy(algos);

ll\_destroy(task->seeders);

free(task);

printf("\a\n");

system("pause");

return true;

}

MENU\_HANDLER(testAlgos) {

struct MenuEntry\*\* entries = calloc(4, sizeof(struct MenuEntry\*));

entries[0] = menu\_entry\_new("Test Select №6", &testSelect6);

entries[1] = menu\_entry\_new("Test Select №8", &testSelect8);

entries[2] = menu\_entry\_new("Test Select+Exchange №3", &testSelect3Exchange);

entries[3] = menu\_entry\_new("OK", &menu\_handler\_exit);

struct TextMenu\* menu = menu\_new(entries, 4);

menu\_start(menu);

return true;

}

\_\_attribute\_\_((flatten)) void measureDefault() {

struct BenchMarkTask\* task = get\_full\_task();

task->size = size;

struct Table\* results = measure(task);

tbl\_print(results);

tbl\_destroy(results);

ll\_destroy(task->algos);

ll\_destroy(task->seeders);

free(task);

printf("\n\n\n");

}

void measureCase1() {

unsigned int N = 15500;

struct BenchMarkTask\* task = get\_full\_task();

task->multiplier = 3;

for(unsigned int i = 2500; i <= 12500; i += 2500) {

printf("## For vector arr[%d]:\n", i);

SETSIZE(size, 1, 1, i);

task->size = size;

struct Table\* results = measure(task);

tbl\_print(results);

tbl\_destroy(results);

printf("\n\n\n");

}

ll\_destroy(task->algos);

ll\_destroy(task->seeders);

free(task);

unsigned int P = 3;

for(int M = 1; M <= 1024; M \*= 2) {

printf("## For arr[%d][%d][%d]:\n", P, M, N);

SETSIZE(size, P, M, N);

measureDefault();

}

}

void measureCase2() {

unsigned int P = 3;

for(int M = 10; M <= 10000; M \*= 10) {

int N = 10000 / (M / 10);

printf("## For arr[%d][%d][%d]:\n", P, M, N);

SETSIZE(size, P, M, N);

measureDefault();

}

}

void measureCase3() {

unsigned int M = 100;

for(int P = 20; P <= 2000; P \*= 10) {

unsigned int N = 2000 / ((P / 2) / 10);

printf("## For arr[%d][%d][%d]:\n", P, M, N);

SETSIZE(size, P, M, N);

measureDefault();

}

}

MENU\_HANDLER(measureEverything) {

printf("# Study case №1: Impact of column length on execution time of different sorting algorithms:\n\n");

measureCase1();

printf("# Study case №2: Impact of incision shape on execution time of different sorting algorithms:\n\n");

measureCase2();

printf("# Study case №3: Impact of keys quantity on execution time of different sorting algorithms:\n\n");

measureCase3();

printf("\a\a\a");

system("pause");

return true;

}

MENU\_HANDLER(help) {

printf("# SDA Coursework\n\n");

printf("This program demonstrates the behaviour of 3 sorting algorithms:\n");

printf("• Selection sort No. 6\n");

printf("• Selection sort No. 8\n");

printf("• Selection/Exchange sort No. 3\n\n");

printf("To test if the implementation of algorithm is correct, pick 'Test...' in main menu and select the algo you'd like to test. ");

printf("You will be prompted to enter values of M and N (arr[1][M][N]).\n");

printf("To measure all algorithms at once, pick 'Measure everything' option in menu and wait for 3 bells.\n\n");

printf("Coursework done by: KV-14 Oleksandr Briukhanov\n\n");

system("pause");

return true;

}

int main() {

srand(time(NULL));

struct MenuEntry\*\* entries = calloc(4, sizeof(struct MenuEntry\*));

entries[0] = menu\_entry\_new("Measure everything", &measureEverything);

entries[1] = menu\_entry\_new("Test...", &testAlgos);

entries[2] = menu\_entry\_new("Help", &help);

entries[3] = menu\_entry\_new("Quit", &menu\_handler\_exit);

struct TextMenu\* menu = menu\_new(entries, 4);

menu\_start(menu);

return 0;

}

LinkedList.h

#ifndef COURSEWORK\_LINKEDLIST\_H

#define COURSEWORK\_LINKEDLIST\_H

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct LinkedListElement {

/\*\*

\* Pervious element, NULL if the element is first.

\*/

struct LinkedListElement\* prev;

/\*\*

\* Next element, NULL if the element is last.

\*/

struct LinkedListElement\* next;

/\*\*

\* Pointer to the data of element.

\*/

void\* data;

};

struct LinkedList {

/\*\*

\* Pointer to the first element.

\*/

struct LinkedListElement\* begin;

/\*\*

\* Pointer to the last element.

\*/

struct LinkedListElement\* end;

/\*\*

\* Amount of elements in list.

\*/

unsigned int size;

};

/\*\*

\* Creates new LinkedList structure with no elements.

\*/

struct LinkedList\* ll\_new();

/\*\*

\* Same as `ll->size`.

\*/

unsigned int ll\_size(struct LinkedList\* ll);

/\*\*

\* Returns true if ll->size is 0.

\*/

bool ll\_empty(struct LinkedList\* ll);

/\*\*

\* Appends element to the list with `data`.

\*/

void ll\_append(struct LinkedList\* ll, void\* data);

/\*\*

\* Deallocates memory dedicated to the list and it's elements.

\* Will also deallocate element values!

\*/

void ll\_destroy(struct LinkedList\* ll);

/\*\*

\* Gets pointer to the data of element on position `index`.

\*/

void\* ll\_get(struct LinkedList\* ll, int index);

/\*\*

\* Sets pointer to the data of element on position `index`.

\*/

void ll\_set(struct LinkedList\* ll, int index, void\* data);

#endif //COURSEWORK\_LINKEDLIST\_H

LinkedList.c

#include "LinkedList.h"

struct LinkedList\* ll\_new() {

struct LinkedList\* ll = (struct LinkedList\*) malloc(sizeof(struct LinkedList));

ll->begin = NULL;

ll->end = NULL;

ll->size = 0;

return ll;

}

unsigned int ll\_size(struct LinkedList\* ll) {

return ll->size;

}

bool ll\_empty(struct LinkedList\* ll) {

return ll\_size(ll) == 0;

}

void ll\_append(struct LinkedList\* ll, void\* data) {

struct LinkedListElement\* element = (struct LinkedListElement\*) malloc(sizeof(struct LinkedListElement));

element->data = data;

element->next = NULL;

if(ll\_empty(ll)) {

element->prev = NULL;

ll->begin = element;

ll->end = element;

} else {

struct LinkedListElement \*last = ll->end;

element->prev = last;

last->next = ll->end = element;

}

ll->size++;

}

void ll\_destroy(struct LinkedList\* ll) {

struct LinkedListElement\* el = ll->end;

while(true) {

struct LinkedListElement\* prev = el->prev;

free(el->data);

free(el);

if(prev == NULL)

break;

else

el = prev;

}

free(ll);

}

/\*\*

\* Same as `ll\_get` but returns pointer to the LinkedListElement structure instead of the data inside it.

\*/

struct LinkedListElement\* priv\_ll\_get\_internal(struct LinkedList\* ll, int index) {

if(index < 0) {

index = ll\_size(ll) + index - 1;

if(index < 0) {

fprintf(stderr, "[LinkedList:priv\_ll\_get\_internal] Index %d is out of range.\n");

return NULL;

}

} else if (index >= ll\_size(ll)) {

fprintf(stderr, "[LinkedList:priv\_ll\_get\_internal] Index %d is out of range.\n");

return NULL;

}

struct LinkedListElement\* el = ll->begin;

for(int i = 0; i < index; i++)

el = el->next;

return el;

}

void\* ll\_get(struct LinkedList\* ll, int index) {

struct LinkedListElement\* el = priv\_ll\_get\_internal(ll, index);

return el->data;

}

void ll\_set(struct LinkedList \*ll, int index, void\* data) {

struct LinkedListElement\* el = priv\_ll\_get\_internal(ll, index);

el->data = data;

}

Table.h

#ifndef COURSEWORK\_TABLE\_H

#define COURSEWORK\_TABLE\_H

#include <stdarg.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include "LinkedList.h"

/\*\*

\* Table structure.

\* Stores the name of table, it's rows and lengths of cells.

\*/

struct Table {

char\* name;

struct LinkedList\* rows;

struct LinkedList\* cellLengths;

};

/\*\*

\* Creates new table with name/heading `name`.

\*/

struct Table\* tbl\_new(char\* name);

/\*\*

\* Returns number of rows.

\*/

unsigned int tbl\_height(struct Table\* t);

/\*\*

\* Returns number of columns.

\*/

unsigned int tbl\_width(struct Table\* t);

/\*\*

\* Adds new row to the table.

\*

\* @param num - Number of cells (passed as arguments)

\*/

void tbl\_append(struct Table\* t, unsigned int num, ...);

/\*\*

\* Imports row to table from (char\*)[] array.

\*/

void tbl\_append\_arr(struct Table\* t, unsigned int num, char\*\* cells);

/\*\*

\* Prints the table to STDOUT.

\*/

void tbl\_print(struct Table\* t);

/\*\*

\* Deallocates memory for table structure, rows and cellLengths linked lists.

\*/

void tbl\_destroy(struct Table\* t);

#endif //COURSEWORK\_TABLE\_H

Table.c

#include "Table.h"

struct Table\* tbl\_new(char\* name) {

struct Table\* table = malloc(sizeof(struct Table));

table->name = name;

table->rows = ll\_new();

table->cellLengths = ll\_new();

return table;

}

unsigned int tbl\_height(struct Table\* t) {

return ll\_size(t->rows);

}

unsigned int tbl\_width(struct Table\* t) {

return ll\_size(t->cellLengths);

}

/\*\*

\* Returns width of the tables in screen characters (not columns).

\*/

unsigned int priv\_tbl\_width\_chars(struct Table\* t) {

unsigned int chars = 0;

unsigned int width = tbl\_width(t);

for(int i = 0; i < width; i++)

chars += \*((unsigned int\*) ll\_get(t->cellLengths, i)) + 2;

return chars + width + 1;

}

void tbl\_append(struct Table\* t, unsigned int num, ...) {

va\_list valist;

va\_start(valist, num);

struct LinkedList\* row = ll\_new();

for(int i = 0; i < num; i++) {

char\* cell = va\_arg(valist, char\*);

unsigned int\* cellSize = malloc(sizeof(unsigned int));

\*cellSize = strlen(cell);

if((i + 1) > ll\_size(t->cellLengths)) {

ll\_append(t->cellLengths, cellSize);

} else {

unsigned int\* currentColumnSize = ll\_get(t->cellLengths, i);

if(\*cellSize > \*currentColumnSize)

ll\_set(t->cellLengths, i, cellSize);

}

ll\_append(row, cell);

}

ll\_append(t->rows, row);

va\_end(valist);

}

void tbl\_append\_arr(struct Table\* t, unsigned int num, char\*\* cells) {

struct LinkedList\* row = ll\_new();

for(int i = 0; i < num; i++) {

char\* cell = cells[i];

unsigned int\* cellSize = malloc(sizeof(unsigned int));

\*cellSize = strlen(cell);

if((i + 1) > ll\_size(t->cellLengths)) {

ll\_append(t->cellLengths, cellSize);

} else {

unsigned int\* currentColumnSize = ll\_get(t->cellLengths, i);

if(\*cellSize > \*currentColumnSize)

ll\_set(t->cellLengths, i, cellSize);

}

ll\_append(row, cell);

}

ll\_append(t->rows, row);

}

/\*\*

\* Prints heading of the table.

\*/

void priv\_tbl\_print\_header(struct Table\* t) {

printf("╔");

unsigned int chars = priv\_tbl\_width\_chars(t) - 2;

for(int i = 0; i < chars; i++)

printf("═");

printf("╗\n║");

unsigned int spaces = (chars - strlen(t->name)) / 2;

for(int i = 0; i < spaces; i++)

putchar(' ');

printf("%s", t->name);

for(int i = 0; i < (spaces + (strlen(t->name) % 2 == 0 ? 1 : 0)); i++)

putchar(' ');

printf("║\n╠");

unsigned int width = tbl\_width(t);

for(int i = 0; i < width; i++) {

unsigned int hyphens = \*((unsigned int\*) ll\_get(t->cellLengths, i)) + 2;

for(int j = 0; j < hyphens; j++)

printf("═");

printf(i == (width - 1) ? "╣\n" : "╦");

}

}

/\*\*

\* Prints a single row.

\*/

void priv\_tbl\_print\_row(struct Table\* t, struct LinkedList\* row, bool closingBottom) {

printf("║");

unsigned int width = tbl\_width(t);

for(int i = 0; i < width; i++) {

char\* cell = (char\*) ll\_get(row, i);

unsigned int spaces = \*((unsigned int\*) ll\_get(t->cellLengths, i)) - strlen(cell) + 1;

for(int j = 0; j < spaces; j++)

putchar(' ');

printf("%s ║", cell);

}

printf("\n%s", closingBottom ? "╚" : "╠");

for(int i = 0; i < width; i++) {

unsigned int hyphens = \*((unsigned int\*) ll\_get(t->cellLengths, i)) + 2;

for(int j = 0; j < hyphens; j++)

printf("═");

if(i < width - 1)

printf(closingBottom ? "╩" : "╬");

}

printf(closingBottom ? "╝" : "╣\n");

}

void tbl\_print(struct Table\* t) {

priv\_tbl\_print\_header(t);

unsigned int height = tbl\_height(t);

for(int i = 0; i < height; i++) {

struct LinkedList\* row = ll\_get(t->rows, i);

priv\_tbl\_print\_row(t, row, i == (height - 1) ? true : false);

}

printf("\n");

}

void tbl\_destroy(struct Table\* t) {

ll\_destroy(t->cellLengths);

ll\_destroy(t->rows);

free(t);

}

Menu.h

#ifndef COURSEWORK\_MENU\_H

#define COURSEWORK\_MENU\_H

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MENU\_HANDLER(name) bool name(void)

/\*\*

\* Menu handler is a function with an arity of 0 that returns:

\* 1) true, if user should be returned back to the menu

\* 2) false, if menu should be closed

\*

\* Pointer to menu handlers is typealiased as MenuEntryHandler

\*/

typedef bool (\*MenuEntryHandler)(void);

/\*\*

\* Menu entry structure, contains it's title (name) and handler (hand).

\*/

struct MenuEntry {

char\* name;

MenuEntryHandler hand;

};

/\*\*

\* Creates new MenuEntry with name `name` and handler `hand`.

\*/

struct MenuEntry\* menu\_entry\_new(char\* name, MenuEntryHandler hand);

/\*\*

\* Menu handler that always returns false (exits menu).

\*/

bool menu\_handler\_exit();

/\*\*

\* Text Menu structure)

\*/

struct TextMenu {

/\*\*

\* Array of pointers to menu entries.

\*/

struct MenuEntry\*\* menuEntries;

/\*\*

\* Stores count of entries.

\*/

unsigned int entryCount;

/\*\*

\* Stores the index of menu entry currently selected.

\*/

unsigned int currentIndex;

/\*\*

\* Is set to negated value of last executed menu entry handler (false by default).

\* If true, next loop will clean the screen and exit instead of printing menu and continuing.

\*/

bool willExit;

/\*\*

\* Header and footer of the menu. "KV-14 Oleksandr Briukhanov" by default. Can't be set via constructor `menu\_new`.

\*/

char\* doneBy;

};

/\*\*

\* Creates new TextMenu from entries (`count` of entries should be provided).

\*/

struct TextMenu\* menu\_new(struct MenuEntry\*\* entries, unsigned int count);

/\*\*

\* Starts the menu.

\* Executes loop which updates state of the menu once user has pressed a button and then prints the menu to STDOUT until user picks an entry,

\* which will execute a handler which will return false.

\*/

int menu\_start(struct TextMenu\* menu);

#endif //COURSEWORK\_MENU\_H

Menu.c

#include "Menu.h"

#include "getch.h"

struct MenuEntry\* menu\_entry\_new(char\* name, MenuEntryHandler hand) {

struct MenuEntry\* menu = malloc(sizeof(struct MenuEntry));

menu->name = name;

menu->hand = hand;

return menu;

}

struct TextMenu\* menu\_new(struct MenuEntry\*\* entries, unsigned int count) {

struct TextMenu\* menu = malloc(sizeof(struct TextMenu));

menu->currentIndex = 0;

menu->willExit = false;

menu->doneBy = "KV-14 Oleksandr Briukhanov";

menu->menuEntries = entries;

menu->entryCount = count;

return menu;

}

bool menu\_handler\_exit() {

return false;

}

/\*\*

\* Executes the action bound to the currently selected menu entry.

\*/

void priv\_menu\_execute\_current\_action(struct TextMenu\* menu) {

struct MenuEntry\* entry = menu->menuEntries[menu->currentIndex];

system("clear");

menu->willExit = !(entry->hand)();

printf("\a\n");

}

/\*\*

\* Prints menu to STDOUT.

\*/

void priv\_menu\_print\_view(struct TextMenu\* menu) {

system("clear");

printf("Coursework done by: %s\n\n", menu->doneBy);

for(int i = 0; i < menu->entryCount; i++)

printf("%c %s\n", i == menu->currentIndex ? '>' : ' ', menu->menuEntries[i]->name);

printf("Coursework done by: %s\n\n", menu->doneBy);

}

int menu\_start(struct TextMenu\* menu) {

while(!menu->willExit) {

priv\_menu\_print\_view(menu);

switch(getch()) {

case '\n':

case '\r':

priv\_menu\_execute\_current\_action(menu);

break;

case '[': ;

int arrow = getch();

if(arrow == 'A') {

menu->currentIndex--;

if(menu->currentIndex >= menu->entryCount) // overflow

menu->currentIndex = menu->entryCount - 1;

} else if(arrow == 'B') {

menu->currentIndex++;

if(menu->currentIndex >= menu->entryCount)

menu->currentIndex = 0;

}

break;

}

}

return 0;

}

getch.h

#ifndef COURSEWORK\_GETCH\_H

#define COURSEWORK\_GETCH\_H

#include <stdio.h>

#include <unistd.h>

#include <termios.h>

/\*\*

\* Returns number of key pressed. conio.h polyfill for Scientific Linux.

\*/

int getch(void);

#endif //COURSEWORK\_GETCH\_H

#include "getch.h"

#include <stdlib.h>

getch.c

int getch(void)

{

int buf = 0;

struct termios old = {0};

fflush(stdout);

if(tcgetattr(0, &old) < 0)

perror("tcsetattr()");

old.c\_lflag &= ~ICANON;

old.c\_lflag &= ~ECHO;

old.c\_cc[VMIN] = 1;

old.c\_cc[VTIME] = 0;

if(tcsetattr(0, TCSANOW, &old) < 0)

perror("tcsetattr ICANON");

if(read(0, &buf, 1) < 0)

perror("read()");

old.c\_lflag |= ICANON;

old.c\_lflag |= ECHO;

if(tcsetattr(0, TCSADRAIN, &old) < 0)

perror("tcsetattr ~ICANON");

return buf;

}

Generators.h

#ifndef COURSEWORK\_GENERATORS\_H

#define COURSEWORK\_GENERATORS\_H

#include <stdlib.h>

#include <time.h>

/\*\*

\* Generators are functions which generate values to fill arrays with.

\* Generators must accept the pervious value (0 if none) and return the next one.

\* Example of filling array with the use of generator:

\*

\* unsigned int perv = 0;

\* for(...)

\* arr[i] = perv = generator(perv);

\*

\* Pointers to the generating functions are typealiased to Generator.

\*/

typedef unsigned int(\*Generator)(unsigned int, unsigned int, unsigned int, unsigned int);

/\*\*

\* ordered 0 = 1

\* ordered n = n + 1

\*/

unsigned int ordered(unsigned int perv, unsigned P, unsigned M, unsigned N);

/\*\*

\* ordered 0 = P\*M\*N

\* ordered n = n - 1

\*/

unsigned int backordered(unsigned int perv, unsigned P, unsigned M, unsigned N);

/\*\*

\* ordered n = [random value between 0 and P\*M\*N]

\*/

unsigned int randomized(unsigned int perv, unsigned P, unsigned M, unsigned N);

#endif //COURSEWORK\_GENERATORS\_H

Generators.c

#include "Generators.h"

unsigned int ordered(unsigned int perv, unsigned P, unsigned M, unsigned N) {

return perv + 1;

}

unsigned int backordered(unsigned int perv, unsigned P, unsigned M, unsigned N) {

if(perv == 0)

return P \* M \* N;

return perv - 1;

}

unsigned int randomized(unsigned int perv, unsigned P, unsigned M, unsigned N) {

unsigned int max = P\*M\*N;

return rand() % (max + 1);

}

Algos.h

ifndef COURSEWORK\_ALGOS\_H

#define COURSEWORK\_ALGOS\_H

#include <time.h>

#include "TestArray.h"

/\*\*

\* Algorithms in this coursework have the same signature so for additional usability and readability of code

\* pointers to them are typealiased to Algo.

\*

\* Algorithms must accept the array as the first argument and size as the next three. Algorithms return the time taken for sorting (clock\_t).

\*/

typedef clock\_t (\*Algo)(TestArray, unsigned int, unsigned int, unsigned int);

/\*\*

\* Due to all of the algorithms having the same signature and performing same operation on start it is

\* possible to emphasize the body of actual algorithms by hiding the initialization and declaraction commands under this

\* handy BEGIN\_ALGO macro. Since this macro defines a function, but doesn't end it's body with a "{" or even return, use

\* END\_ALGO right after the algorithm body.

\*/

#define BEGIN\_ALGO(name) \

clock\_t name(TestArray arr, unsigned int P, unsigned int M, unsigned int N) { \

clock\_t time\_start = clock(), time\_stop;

/\*\*

\* Ends algorithm definition with return statemend and the closing brace.

\*/

#define END\_ALGO \

time\_stop = clock(); \

return time\_stop - time\_start; \

}

/\*\*

\* Defines vector Sum with the sums of columns in the current incision of the array.

\* For use in the first loop of algorithms.

\*/

#define INIT\_SUM \

unsigned int\* Sum = calloc(N, sizeof(unsigned int)); \

for(unsigned int j = 0; j < N; j++) { \

Sum[j] = 0; \

for(unsigned int k = 0; k < M; k++) \

Sum[j] += arr[p][k][j]; \

}

/\*\*

\* Due to all of the algorithms requirement to swap elements in the same fashion (swap in Sum vector and swap each element of columns with same indices),

\* it is possible to get rid of redundant noise in algorithm bodies (to improve the readability of code) by hiding this operation under SWAP\_ELEMENTS macro.

\*

\* For use anywhere in alogirithms after Sum has been declared.

\*/

#define SWAP\_ELEMENTS(a, b) \

unsigned int temp = Sum[a]; \

Sum[a] = Sum[b]; \

Sum[b] = temp; \

for(unsigned int sort\_k = 0; sort\_k < M; sort\_k++) { \

unsigned int buf = arr[p][sort\_k][a]; \

arr[p][sort\_k][a] = arr[p][sort\_k][b]; \

arr[p][sort\_k][b] = buf; \

}

/\*\*

\* Same as SWAP\_ELEMENTS but with a confition.

\*/

#define SWAP\_ELEMENTS\_IF(condition, a, b) \

if(condition) { \

SWAP\_ELEMENTS(a, b) \

}

/\*\*

\* Selection sort number six.

\*/

clock\_t Select6(TestArray arr, unsigned int P, unsigned int M, unsigned int N);

/\*\*

\* Selection sort number eight.

\*/

clock\_t Select8(TestArray arr, unsigned int P, unsigned int M, unsigned int N);

/\*\*

\* Selection and exchange sort hybrid number three.

\*/

clock\_t Select3Exchange(TestArray arr, unsigned int P, unsigned int M, unsigned int N);

#endif //COURSEWORK\_ALGOS\_H

Algos.c

#include "Algos.h"

BEGIN\_ALGO(Select6)

for(unsigned int p = 0; p < P; p++) {

INIT\_SUM;

unsigned int min;

for(unsigned int i = 0; i < N - 1; i++) {

min = i;

for(unsigned int j = i + 1; j < N; j++)

if(Sum[j] < Sum[min])

min = j;

SWAP\_ELEMENTS\_IF(min != i, min, i);

}

free(Sum);

}

END\_ALGO

BEGIN\_ALGO(Select8)

for(unsigned int p = 0; p < P; p++) {

INIT\_SUM;

int L = 0, R = N - 1;

while(L < R) {

int min = L, max = L;

for(int i = L + 1; i < R + 1; i++) {

if(Sum[i] < Sum[min])

min = i;

else if(Sum[i] > Sum[max])

max = i;

}

SWAP\_ELEMENTS\_IF(min != L, min, L);

if(max != R) {

SWAP\_ELEMENTS\_IF(max == L, min, R);

SWAP\_ELEMENTS\_IF(max != L, max, R);

}

L++;

R--;

}

free(Sum);

}

END\_ALGO

BEGIN\_ALGO(Select3Exchange)

for(unsigned int p = 0; p < P; p++) {

INIT\_SUM;

int L = 0, R = N - 1;

while(L < R) {

SWAP\_ELEMENTS\_IF(Sum[L] > Sum[R], L, R);

unsigned int min = Sum[L], max = Sum[R];

for(int i = L + 1; i < R + 1; i++) {

if(Sum[i] < min) {

min = Sum[i];

SWAP\_ELEMENTS(i, L);

} else if(Sum[i] > max) {

max = Sum[i];

SWAP\_ELEMENTS(i, R);

}

}

L++;

R--;

}

free(Sum);

}

END\_ALGO

TestArray.h

#ifndef COURSEWORK\_TESTARRAY\_H

#define COURSEWORK\_TESTARRAY\_H

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include "Generators.h"

typedef unsigned int\*\*\* TestArray;

/\*\*

\* Creates new test 3D array, using seed to generate it's values.

\* Will print errors to stdout and return NULL if operation can't be completed (memory poisoned, not enough memory).

\*/

TestArray ta\_new(Generator seed, unsigned P, unsigned M, unsigned N);

/\*\*

\* Prints contents of array's incisions.

\*/

void ta\_describe(TestArray arr, unsigned P, unsigned M, unsigned N);

/\*\*

\* Deallocates memory for array arr and sets \*arr to NULL.

\*/

void ta\_destroy(TestArray\* arr, unsigned P, unsigned M, unsigned N);

#endif //COURSEWORK\_TESTARRAY\_H

TestArray.c

#include "TestArray.h"

TestArray ta\_new(Generator seed, unsigned int P, unsigned int M, unsigned int N) {

unsigned int perv = 0;

TestArray arr = calloc(P, sizeof(unsigned int\*\*));

for(unsigned int p = 0; p < P; p++) {

arr[p] = calloc(M, sizeof(unsigned int\*));

for(unsigned int m = 0; m < M; m++) {

arr[p][m] = calloc(N, sizeof(unsigned int));

if(arr[p][m] == NULL) {

if(errno == ENOMEM)

fprintf(stderr, "E: Not enough memory to create array!\n");

else

fprintf(stderr, "E: Failed to allocate %zu bytes: Fault %d\n", N \* sizeof(unsigned int), errno);

abort();

}

for(unsigned int n = 0; n < N; n++) {

perv = seed(perv, P, M, N);

arr[p][m][n] = perv;

}

}

}

return arr;

}

void ta\_describe(TestArray arr, unsigned int P, unsigned int M, unsigned int N) {

for(unsigned int p = 0; p < P; p++) {

printf("===\nIncision №%d out of %d:\n\n", p + 1, P);

for(unsigned int m = 0; m < M; m++) {

for(unsigned int n = 0; n < N; n++) {

printf("%d", arr[p][m][n]);

printf(n == N - 1 ? ";" : ", ");

}

printf("\n");

}

printf("\nSum:\n");

for(int n = 0; n < N; n++) {

unsigned int sum = 0;

for(int m = 0; m < M; m++)

sum += arr[p][m][n];

printf("%d", sum);

printf(n == N - 1 ? ";\n===\n\n" : ", ");

}

}

}

void ta\_destroy(TestArray\* arr, unsigned int P, unsigned int M, unsigned int N) {

for(unsigned int p = 0; p < P; p++) {

for(unsigned int m = 0; m < M; m++)

free((\*arr)[p][m]);

free((\*arr)[p]);

}

free(\*arr);

\*arr = NULL;

}

BenchMark.h

#ifndef COURSEWORK\_BENCHMARK\_H

#define COURSEWORK\_BENCHMARK\_H

#include "LinkedList.h"

#include "Algos.h"

#define REJECT 2.0

#define MINMAX 3.0

#define SETSIZE(var, p, m, n) \

var = calloc(3, sizeof(unsigned int)); \

var[0] = p; \

var[1] = m; \

var[2] = n;

/\*\*

\* Structure that stores pointer to algorithm and it's name in zero terminated string.

\*/

struct NamedAlgo {

char\* name;

Algo algo;

};

/\*\*

\* Structure that stores pointer to generator and it's name in zero terminated string.

\*/

struct NamedGenerator {

char\* name;

Generator generator;

};

/\*\*

\* Description of measurement task for measure() to execute.

\*/

struct BenchMarkTask {

struct LinkedList\* algos; // LinkedList<NamedAlgo>

struct LinkedList\* seeders; // LinkedList<NamedGenerator>

unsigned int multiplier; // Multiply calculated time by N

unsigned int iterations; // Execute measurements N times

unsigned int\* size; // {P, M, N} array

bool describe; // Call ta\_describe() before and after sort?

};

/\*\*

\* Shorthand for

\* NamedAlgo\* x = malloc(...);

\* x->name = name;

\* x->algo = algo;

\*/

struct NamedAlgo\* na\_new(char\* name, Algo algo);

/\*\*

\* Shorthand for

\* NamedGenerator\* x = malloc(...);

\* x->name = name;

\* x->generator = generator;

\*/

struct NamedGenerator\* ng\_new(char\* name, Generator generator);

/\*\*

\* Shorthand for defining NamedAlgo which is named the same way as it's function in source.

\*/

#define ALGO(name) na\_new(#name, &name)

/\*\*

\* Shorthand for defining NamedGenerator which is named the same way as it's function in source.

\*/

#define SEED(name) ng\_new(#name, &name)

/\*\*

\* Executes the measurement task and returns table with results ready to be printed.

\*/

struct Table\* measure(struct BenchMarkTask\* task);

#endif //COURSEWORK\_BENCHMARK\_H

BenchMark.c

#include "BenchMark.h"

#include "Table.h"

#define ARRSIZE task->size[0], task->size[1], task->size[2]

struct NamedAlgo\* na\_new(char\* name, Algo algo) {

struct NamedAlgo\* res = malloc(sizeof(struct NamedAlgo));

res->name = name;

res->algo = algo;

return res;

}

struct NamedGenerator\* ng\_new(char\* name, Generator generator) {

struct NamedGenerator\* res = malloc(sizeof(struct NamedGenerator));

res->name = name;

res->generator = generator;

return res;

}

/\*\*

\* Compute mean time by the way described in the coursework taskbook.

\*/

long double priv\_bc\_time(unsigned int iterations, clock\_t\* measurements) {

clock\_t buf;

int L = REJECT, R = iterations - 1;

int k = REJECT;

for(int j = 0; j < MINMAX; j++) {

for(int i = L; i < R; i++) {

if(measurements[i] > measurements[i + 1]) {

buf = measurements[i];

measurements[i] = measurements[i + 1];

measurements[i + 1] = buf;

}

}

R = k;

for(int i = R - 1; i >= L; i--) {

if(measurements[i] > measurements[i + 1]) {

buf = measurements[i];

measurements[i] = measurements[i + 1];

measurements[i + 1] = buf;

k = i;

}

}

L = k + 1;

}

clock\_t sum = 0;

for(int i = REJECT + MINMAX; i < iterations - MINMAX; i++)

sum += measurements[i];

return (long double) (sum / (iterations - 2\*MINMAX - REJECT));

}

struct Table\* measure(struct BenchMarkTask\* task) {

if(task->iterations < 12) {

fprintf(stderr, "E: Invalid iterations param: at least 12 expected, %d found\n", task->iterations);

return NULL;

}

struct Table\* table = tbl\_new("Measurements");

unsigned int seedersCount = ll\_size(task->seeders);

char\*\* firstRow = calloc(seedersCount + 1, sizeof(char\*));

firstRow[0] = "Algo / Seed";

for(int i = 0; i < seedersCount; i++)

firstRow[i + 1] = ((struct NamedGenerator\*) ll\_get(task->seeders, i))->name;

tbl\_append\_arr(table, seedersCount + 1, firstRow);

unsigned int algoCount = ll\_size(task->algos);

for(int i = 0; i < algoCount; i++) {

// For each algo:

struct NamedAlgo\* algo = ll\_get(task->algos, i);

char\*\* row = calloc(seedersCount + 1, sizeof(char\*));

row[0] = algo->name;

// Execute tests for each generator:

for(int j = 0; j < seedersCount; j++) {

struct NamedGenerator\* seeder = ll\_get(task->seeders, j);

// Initialize measurements array

clock\_t\* measurements = calloc(task->iterations, sizeof(clock\_t));

for(int k = 0; k < task->iterations; k++) {

// For each iteration of tests:

// 1. Create test array.

TestArray array = ta\_new(seeder->generator, ARRSIZE);

// 2. Sort array:

// 2.1. If task requires debug printing, call ta\_describe before and after the sort on the last iteration.

if(k == task->iterations - 1 && task->describe) {

printf("Before sort:\n");

ta\_describe(array, ARRSIZE);

measurements[k] = algo->algo(array, ARRSIZE);

printf("After sort:\n");

ta\_describe(array, ARRSIZE);

} else {

// 2.2. Else, just sort and store the time in measurements.

measurements[k] = algo->algo(array, ARRSIZE);

}

// 3. Deallocate array

ta\_destroy(&array, ARRSIZE);

}

char\* time = malloc(256);

sprintf(time, "%Lf%c", priv\_bc\_time(task->iterations, measurements) \* task->multiplier, '\0'); // Compute mean time

row[j + 1] = time;

}

tbl\_append\_arr(table, seedersCount + 1, row);

}

return table;

}

main.c

#include "LinkedList.h"

#include "BenchMark.h"

#include "Table.h"

#include "Menu.h"

unsigned int\* size;

/\*\*

\* Create default measurement task which includes all generators (ordered, randomized, backordered).

\* Description is turned off and the amount of iterations is reduced to 12. Algo list is undefined.

\*/

\_\_attribute\_\_((flatten)) struct BenchMarkTask\* get\_default\_task() {

struct LinkedList\* seeds = ll\_new();

ll\_append(seeds, SEED(ordered));

ll\_append(seeds, SEED(randomized));

ll\_append(seeds, SEED(backordered));

struct BenchMarkTask\* task = malloc(sizeof(struct BenchMarkTask));

task->seeders = seeds;

task->describe = false;

task->multiplier = 1;

task->iterations = 12;

return task;

}

/\*\*

\* Create full measurement task which includes all generators and all algorithms.

\* Description is turned off, but the amount of iterations is set to 29.

\*/

\_\_attribute\_\_((flatten)) struct BenchMarkTask\* get\_full\_task() {

struct LinkedList\* algos = ll\_new();

ll\_append(algos, ALGO(Select6));

ll\_append(algos, ALGO(Select8));

ll\_append(algos, ALGO(Select3Exchange));

struct BenchMarkTask\* task = get\_default\_task();

task->algos = algos;

task->iterations = 28;

return task;

}

// Test Select6 (get\_default\_task+description+Select6)

MENU\_HANDLER(testSelect6) {

struct LinkedList\* algos = ll\_new();

ll\_append(algos, ALGO(Select6));

SETSIZE(size, 1, 0, 0);

printf("P = const = 1\nM = ");

fflush(stdout);

scanf("%d", &size[1]);

printf("N = ");

fflush(stdout);

scanf("%d", &size[2]);

struct BenchMarkTask\* task = get\_default\_task();

task->algos = algos;

task->describe = true;

task->size = size;

struct Table\* results = measure(task);

tbl\_print(results);

tbl\_destroy(results);

ll\_destroy(algos);

ll\_destroy(task->seeders);

free(task);

printf("\a\n");

system("pause");

return true;

}

// Test Select8 (get\_default\_task+description+Select8)

MENU\_HANDLER(testSelect8) {

struct LinkedList\* algos = ll\_new();

ll\_append(algos, ALGO(Select8));

SETSIZE(size, 1, 0, 0);

printf("P = const = 1\nM = ");

fflush(stdout);

scanf("%d", &size[1]);

printf("N = ");

fflush(stdout);

scanf("%d", &size[2]);

struct BenchMarkTask\* task = get\_default\_task();

task->algos = algos;

task->describe = true;

task->size = size;

struct Table\* results = measure(task);

tbl\_print(results);

tbl\_destroy(results);

ll\_destroy(algos);

ll\_destroy(task->seeders);

free(task);

printf("\a\n");

system("pause");

return true;

}

// Test testSelect3Exchange (get\_default\_task+description+testSelect3Exchange)

MENU\_HANDLER(testSelect3Exchange) {

struct LinkedList\* algos = ll\_new();

ll\_append(algos, ALGO(Select3Exchange));

SETSIZE(size, 1, 0, 0);

printf("P = const = 1\nM = ");

fflush(stdout);

scanf("%d", &size[1]);

printf("N = ");

fflush(stdout);

scanf("%d", &size[2]);

struct BenchMarkTask\* task = get\_default\_task();

task->algos = algos;

task->describe = true;

task->size = size;

struct Table\* results = measure(task);

tbl\_print(results);

tbl\_destroy(results);

ll\_destroy(algos);

ll\_destroy(task->seeders);

free(task);

printf("\a\n");

system("pause");

return true;

}

// Algo test menu

MENU\_HANDLER(testAlgos) {

struct MenuEntry\*\* entries = calloc(4, sizeof(struct MenuEntry\*));

entries[0] = menu\_entry\_new("Test Select №6", &testSelect6);

entries[1] = menu\_entry\_new("Test Select №8", &testSelect8);

entries[2] = menu\_entry\_new("Test Select+Exchange №3", &testSelect3Exchange);

entries[3] = menu\_entry\_new("OK", &menu\_handler\_exit);

struct TextMenu\* menu = menu\_new(entries, 4);

menu\_start(menu);

return true;

}

// Executes get\_full\_task with size declared on line 6 (can be set with SETSIZE) and prints the results (will deallocate memory afterwards).

\_\_attribute\_\_((flatten)) void measureDefault() {

struct BenchMarkTask\* task = get\_full\_task();

task->size = size;

struct Table\* results = measure(task);

tbl\_print(results);

tbl\_destroy(results);

ll\_destroy(task->algos);

ll\_destroy(task->seeders);

free(task);

printf("\n\n\n");

}

void measureCase1() {

unsigned int N = 15500;

printf("## For vector %d:\n", N);

SETSIZE(size, 1, 1, N);

struct BenchMarkTask\* task = get\_full\_task();

task->size = size;

task->multiplier = 3;

struct Table\* results = measure(task);

tbl\_print(results);

tbl\_destroy(results);

ll\_destroy(task->algos);

ll\_destroy(task->seeders);

free(task);

printf("\n\n\n");

unsigned int P = 3;

for(int M = 1; M <= 1024; M \*= 2) {

printf("## For arr[%d][%d][%d]:\n", P, M, N);

SETSIZE(size, P, M, N);

measureDefault();

}

}

void measureCase2() {

unsigned int P = 3;

for(int M = 10; M <= 10000; M \*= 10) {

int N = 10000 / (M / 10);

printf("## For arr[%d][%d][%d]:\n", P, M, N);

SETSIZE(size, P, M, N);

measureDefault();

}

}

void measureCase3() {

unsigned int M = 100;

for(int P = 20; P <= 2000; P \*= 10) {

unsigned int N = 2000 / ((P / 2) / 10);

printf("## For arr[%d][%d][%d]:\n", P, M, N);

SETSIZE(size, P, M, N);

measureDefault();

}

}

MENU\_HANDLER(measureEverything) {

printf("# Study case №1: Impact of column length on execution time of different sorting algorithms:\n\n");

measureCase1();

printf("# Study case №2: Impact of incision shape on execution time of different sorting algorithms:\n\n");

measureCase2();

printf("# Study case №3: Impact of keys quantity on execution time of different sorting algorithms:\n\n");

measureCase3();

printf("\a\a\a");

system("pause");

return true;

}

MENU\_HANDLER(help) {

printf("# SDA Coursework\n\n");

printf("This program demonstrates the behaviour of 3 sorting algorithms:\n");

printf("• Selection sort No. 6\n");

printf("• Selection sort No. 8\n");

printf("• Selection/Exchange sort No. 3\n\n");

printf("To test if the implementation of algorithm is correct, pick 'Test...' in main menu and select the algo you'd like to test. ");

printf("You will be prompted to enter values of M and N (arr[1][M][N]).\n");

printf("To measure all algorithms at once, pick 'Measure everything' option in menu and wait for 3 bells.\n\n");

printf("Coursework done by: KV-14 Oleksandr Briukhanov\n\n");

system("pause");

return true;

}

int main() {

srand(time(NULL));

struct MenuEntry\*\* entries = calloc(4, sizeof(struct MenuEntry\*));

entries[0] = menu\_entry\_new("Measure everything", &measureEverything);

entries[1] = menu\_entry\_new("Test...", &testAlgos);

entries[2] = menu\_entry\_new("Help", &help);

entries[3] = menu\_entry\_new("Quit", &menu\_handler\_exit);

struct TextMenu\* menu = menu\_new(entries, 4);

menu\_start(menu);

return 0;

}

CmakeLists.txt

cmake\_minimum\_required(VERSION 2.8)

project(coursework C)

set(CMAKE\_C\_STANDARD 99)

add\_compile\_options(-std=gnu99 -O0)

add\_executable(coursework main.c getch.c getch.h Table.c LinkedList.c Table.h LinkedList.h Menu.h Menu.c Algos.c Algos.h TestArray.c TestArray.h Generators.c Generators.h BenchMark.c BenchMark.h)

Тести роботи програми

Select6

P = const = 1

M = 1

N = 10

Before sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

Sum:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

===

After sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

Sum:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

===

Before sort:

===

Incision №1 out of 1:

2, 1, 9, 4, 9, 6, 9, 8, 3, 6;

Sum:

2, 1, 9, 4, 9, 6, 9, 8, 3, 6;

===

After sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 6, 6, 8, 9, 9, 9;

Sum:

1, 2, 3, 4, 6, 6, 8, 9, 9, 9;

===

Before sort:

===

Incision №1 out of 1:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1;

Sum:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1;

===

After sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

Sum:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

===

P = const = 1

M = 5

N = 10

Before sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

11, 12, 13, 14, 15, 16, 17, 18, 19, 20;

21, 22, 23, 24, 25, 26, 27, 28, 29, 30;

31, 32, 33, 34, 35, 36, 37, 38, 39, 40;

41, 42, 43, 44, 45, 46, 47, 48, 49, 50;

Sum:

105, 110, 115, 120, 125, 130, 135, 140, 145, 150;

===

After sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

11, 12, 13, 14, 15, 16, 17, 18, 19, 20;

21, 22, 23, 24, 25, 26, 27, 28, 29, 30;

31, 32, 33, 34, 35, 36, 37, 38, 39, 40;

41, 42, 43, 44, 45, 46, 47, 48, 49, 50;

Sum:

105, 110, 115, 120, 125, 130, 135, 140, 145, 150;

===

Before sort:

===

Incision №1 out of 1:

5, 33, 38, 25, 0, 37, 8, 36, 35, 31;

9, 39, 42, 2, 44, 28, 35, 30, 35, 40;

14, 9, 33, 10, 9, 21, 47, 32, 37, 46;

14, 42, 2, 26, 41, 28, 13, 49, 38, 48;

3, 47, 11, 19, 24, 4, 48, 8, 8, 32;

Sum:

45, 170, 126, 82, 118, 118, 151, 155, 153, 197;

===

After sort:

===

Incision №1 out of 1:

5, 25, 0, 37, 38, 8, 35, 36, 33, 31;

9, 2, 44, 28, 42, 35, 35, 30, 39, 40;

14, 10, 9, 21, 33, 47, 37, 32, 9, 46;

14, 26, 41, 28, 2, 13, 38, 49, 42, 48;

3, 19, 24, 4, 11, 48, 8, 8, 47, 32;

Sum:

45, 82, 118, 118, 126, 151, 153, 155, 170, 197;

===

Before sort:

===

Incision №1 out of 1:

50, 49, 48, 47, 46, 45, 44, 43, 42, 41;

40, 39, 38, 37, 36, 35, 34, 33, 32, 31;

30, 29, 28, 27, 26, 25, 24, 23, 22, 21;

20, 19, 18, 17, 16, 15, 14, 13, 12, 11;

10, 9, 8, 7, 6, 5, 4, 3, 2, 1;

Sum:

150, 145, 140, 135, 130, 125, 120, 115, 110, 105;

===

After sort:

===

Incision №1 out of 1:

41, 42, 43, 44, 45, 46, 47, 48, 49, 50;

31, 32, 33, 34, 35, 36, 37, 38, 39, 40;

21, 22, 23, 24, 25, 26, 27, 28, 29, 30;

11, 12, 13, 14, 15, 16, 17, 18, 19, 20;

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

Sum:

105, 110, 115, 120, 125, 130, 135, 140, 145, 150;

===

Select3Exchange

P = const = 1

M = 1

N = 10

Before sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

Sum:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

===

After sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

Sum:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

===

Before sort:

===

Incision №1 out of 1:

4, 6, 6, 6, 6, 7, 7, 2, 8, 9;

Sum:

4, 6, 6, 6, 6, 7, 7, 2, 8, 9;

===

After sort:

===

Incision №1 out of 1:

2, 4, 6, 6, 6, 6, 7, 7, 8, 9;

Sum:

2, 4, 6, 6, 6, 6, 7, 7, 8, 9;

===

Before sort:

===

Incision №1 out of 1:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1;

Sum:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1;

===

After sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

Sum:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

===

P = const = 1

M = 5

N = 10

Before sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

11, 12, 13, 14, 15, 16, 17, 18, 19, 20;

21, 22, 23, 24, 25, 26, 27, 28, 29, 30;

31, 32, 33, 34, 35, 36, 37, 38, 39, 40;

41, 42, 43, 44, 45, 46, 47, 48, 49, 50;

Sum:

105, 110, 115, 120, 125, 130, 135, 140, 145, 150;

===

After sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

11, 12, 13, 14, 15, 16, 17, 18, 19, 20;

21, 22, 23, 24, 25, 26, 27, 28, 29, 30;

31, 32, 33, 34, 35, 36, 37, 38, 39, 40;

41, 42, 43, 44, 45, 46, 47, 48, 49, 50;

Sum:

105, 110, 115, 120, 125, 130, 135, 140, 145, 150;

===

Before sort:

===

Incision №1 out of 1:

47, 32, 32, 21, 46, 35, 15, 33, 3, 24;

16, 17, 10, 19, 9, 12, 6, 17, 39, 0;

5, 14, 0, 23, 3, 21, 16, 37, 26, 16;

15, 22, 23, 47, 17, 43, 5, 32, 0, 8;

31, 41, 26, 15, 9, 9, 1, 40, 26, 14;

Sum:

114, 126, 91, 125, 84, 120, 43, 159, 94, 62;

===

After sort:

===

Incision №1 out of 1:

15, 24, 46, 32, 3, 47, 35, 21, 32, 33;

6, 0, 9, 10, 39, 16, 12, 19, 17, 17;

16, 16, 3, 0, 26, 5, 21, 23, 14, 37;

5, 8, 17, 23, 0, 15, 43, 47, 22, 32;

1, 14, 9, 26, 26, 31, 9, 15, 41, 40;

Sum:

43, 62, 84, 91, 94, 114, 120, 125, 126, 159;

===

Before sort:

===

Incision №1 out of 1:

50, 49, 48, 47, 46, 45, 44, 43, 42, 41;

40, 39, 38, 37, 36, 35, 34, 33, 32, 31;

30, 29, 28, 27, 26, 25, 24, 23, 22, 21;

20, 19, 18, 17, 16, 15, 14, 13, 12, 11;

10, 9, 8, 7, 6, 5, 4, 3, 2, 1;

Sum:

150, 145, 140, 135, 130, 125, 120, 115, 110, 105;

===

After sort:

===

Incision №1 out of 1:

41, 42, 43, 44, 45, 46, 47, 48, 49, 50;

31, 32, 33, 34, 35, 36, 37, 38, 39, 40;

21, 22, 23, 24, 25, 26, 27, 28, 29, 30;

11, 12, 13, 14, 15, 16, 17, 18, 19, 20;

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

Sum:

105, 110, 115, 120, 125, 130, 135, 140, 145, 150;

===

Select8

P = const = 1

M = 1

N = 10

Before sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

Sum:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

===

After sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

Sum:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

===

Before sort:

===

Incision №1 out of 1:

9, 2, 10, 8, 6, 7, 0, 10, 4, 7;

Sum:

9, 2, 10, 8, 6, 7, 0, 10, 4, 7;

===

After sort:

===

Incision №1 out of 1:

0, 2, 4, 6, 7, 7, 8, 9, 10, 10;

Sum:

0, 2, 4, 6, 7, 7, 8, 9, 10, 10;

===

Before sort:

===

Incision №1 out of 1:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1;

Sum:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1;

===

After sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

Sum:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

===

P = const = 1

M = 5

N = 10

Before sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

11, 12, 13, 14, 15, 16, 17, 18, 19, 20;

21, 22, 23, 24, 25, 26, 27, 28, 29, 30;

31, 32, 33, 34, 35, 36, 37, 38, 39, 40;

41, 42, 43, 44, 45, 46, 47, 48, 49, 50;

Sum:

105, 110, 115, 120, 125, 130, 135, 140, 145, 150;

===

After sort:

===

Incision №1 out of 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

11, 12, 13, 14, 15, 16, 17, 18, 19, 20;

21, 22, 23, 24, 25, 26, 27, 28, 29, 30;

31, 32, 33, 34, 35, 36, 37, 38, 39, 40;

41, 42, 43, 44, 45, 46, 47, 48, 49, 50;

Sum:

105, 110, 115, 120, 125, 130, 135, 140, 145, 150;

===

Before sort:

===

Incision №1 out of 1:

38, 31, 34, 26, 49, 35, 6, 34, 0, 17;

32, 42, 2, 17, 9, 33, 2, 4, 48, 9;

41, 9, 7, 44, 30, 15, 34, 21, 5, 2;

12, 18, 33, 20, 44, 6, 4, 24, 40, 29;

16, 46, 21, 43, 12, 30, 25, 39, 8, 23;

Sum:

139, 146, 97, 150, 144, 119, 71, 122, 101, 80;

===

After sort:

===

Incision №1 out of 1:

6, 17, 34, 0, 35, 34, 38, 49, 31, 26;

2, 9, 2, 48, 33, 4, 32, 9, 42, 17;

34, 2, 7, 5, 15, 21, 41, 30, 9, 44;

4, 29, 33, 40, 6, 24, 12, 44, 18, 20;

25, 23, 21, 8, 30, 39, 16, 12, 46, 43;

Sum:

71, 80, 97, 101, 119, 122, 139, 144, 146, 150;

===

Before sort:

===

Incision №1 out of 1:

50, 49, 48, 47, 46, 45, 44, 43, 42, 41;

40, 39, 38, 37, 36, 35, 34, 33, 32, 31;

30, 29, 28, 27, 26, 25, 24, 23, 22, 21;

20, 19, 18, 17, 16, 15, 14, 13, 12, 11;

10, 9, 8, 7, 6, 5, 4, 3, 2, 1;

Sum:

150, 145, 140, 135, 130, 125, 120, 115, 110, 105;

===

After sort:

===

Incision №1 out of 1:

41, 42, 43, 44, 45, 46, 47, 48, 49, 50;

31, 32, 33, 34, 35, 36, 37, 38, 39, 40;

21, 22, 23, 24, 25, 26, 27, 28, 29, 30;

11, 12, 13, 14, 15, 16, 17, 18, 19, 20;

1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

Sum:

105, 110, 115, 120, 125, 130, 135, 140, 145, 150;

===

Тести вище демонструють правильність роботи алгоритмів на масивах різної степені відсортованості.

Результати роботи програми

Вимірювання швидкодії алгоритмів було зроблено на електронній обчислювальній машині із наступними характеристиками:

|  |  |
| --- | --- |
|  | |
| Операційна система | Scientific Linux 7 (Linux 3.10) |
| Процесор | Intel Xeon E5-2680 2.7GHz |
| Оперативна пам’ять | 993MB DDR3 ECC |
| Компілятор | gcc version 4.8.5 20150623 (Red Hat 4.8.5-44) (GCC) |
| Параметри компілятора | --std=gnu99 -O0 |
| Точність системного часу | 1GHz, CLOCKS\_PER\_SEC= 1000000 |

Виміри були зроблені 28 разів, у таблицях подано лише усереднені за запропонованою методикою дані.

Залежність часу роботи алгоритмів від довжини стовпчиків масива

P = const = 3, M = var, N = const = 15500.

***1) Таблиця 1.0, Вектор***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 1032000 | 1039500 | 955500 |
| Select8 | 663000 | 627000 | 450000 |
| Select3Exchange | 462000 | 1477500 | 462000 |

***2) Таблиця 1.1, M=1***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 893500 | 900000 | 896000 |
| Select8 | 650000 | 614000 | 451500 |
| Select3Exchange | 452000 | 1704500 | 453000 |

***3) Таблиця 1.2, M=2***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 894500 | 907000 | 888000 |
| Select8 | 650000 | 601000 | 450000 |
| Select3Exchange | 451000 | 2204000 | 448500 |

***4) Таблиця 1.3, M=4***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 892000 | 890500 | 894000 |
| Select8 | 643500 | 611000 | 446000 |
| Select3Exchange | 444500 | 3374000 | 458500 |

***5) Таблиця 1.4, M=8***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 905500 | 920000 | 899500 |
| Select8 | 653500 | 625500 | 474500 |
| Select3Exchange | 478500 | 5634000 | 471000 |

***6) Таблиця 1.5, M=16***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 905000 | 936500 | 906000 |
| Select8 | 659500 | 633500 | 459500 |
| Select3Exchange | 475500 | 10412500 | 485500 |

***7) Таблиця 1.6, M=32***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 925000 | 963500 | 926500 |
| Select8 | 656500 | 659500 | 482500 |
| Select3Exchange | 464500 | 20823500 | 520500 |

***8) Таблиця 1.7, M=64***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 921000 | 1014500 | 940000 |
| Select8 | 676000 | 696500 | 503500 |
| Select3Exchange | 477500 | 62488000 | 527500 |

***9) Таблиця 1.8, M=128***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 944500 | 1133500 | 981000 |
| Select8 | 701500 | 866500 | 580000 |
| Select3Exchange | 485500 | 178039000 | 558000 |

***10) Таблиця 1.9, M=256***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 965000 | 1489000 | 1153000 |
| Select8 | 723000 | 1242500 | 770500 |
| Select3Exchange | 547000 | 499721500 | 737500 |

***11) Таблиця 1.10 M=512***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 1308000 | 2691500 | 1978000 |
| Select8 | 1011500 | 2325500 | 1682500 |
| Select3Exchange | 841000 | 1035444500 | 1264500 |

Залежність часу роботи алгоритмів від форми перерізів масива

P = const = 3, M = var, N = var, M\*N = 100000

***1) Таблиця 2.0, M = 10, N = 10000***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 426500 | 446000 | 400000 |
| Select8 | 271000 | 264000 | 188000 |
| Select3Exchange | 217000 | 2754500 | 215500 |

***2) Таблиця 2.1, M = 100, N = 1000***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 5000 | 10500 | 5000 |
| Select8 | 4500 | 6500 | 6500 |
| Select3Exchange | 4500 | 268000 | 5500 |

***3) Таблиця 2.2, M = 1000, N = 100***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 1000 | 2500 | 3500 |
| Select8 | 2500 | 3000 | 3500 |
| Select3Exchange | 2000 | 29500 | 4500 |

***4) Таблиця 2.3, M = 10000, N = 10***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 500 | 5500 | 4500 |
| Select8 | 1500 | 2500 | 4500 |
| Select3Exchange | 2000 | 6000 | 3000 |

Залежність часу роботи алгоритмів від кількості ключів у кожному перерізі масива при однаковій загальній кількості ключів у всьому масиві

P = var, M = const = 100, N = var, P\*N = 40000

***1) Таблиця 3.0, P = 20, N = 2000***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 142500 | 188500 | 156500 |
| Select8 | 99500 | 147000 | 118000 |
| Select3Exchange | 80000 | 6948000 | 105500 |

***2) Таблиця 3.1, P = 200, N = 200***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 30500 | 64000 | 42500 |
| Select8 | 29500 | 64000 | 60500 |
| Select3Exchange | 23000 | 582000 | 39000 |

***2) Таблиця 3.2, P = 2000, N = 20***

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 18500 | 47000 | 32500 |
| Select8 | 19000 | 48500 | 49500 |
| Select3Exchange | 17000 | 85500 | 33000 |

Залежність часу роботи алгоритмів від кількості ключів у одномірному масиві

N = var

**1) Таблиця 4.0, N = 2500**

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 30000 | 30000 | 27000 |
| Select8 | 15000 | 15000 | 12000 |
| Select3Exchange | 15000 | 36000 | 13500 |

**2) Таблиця 4.1, N = 5000**

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 108000 | 108000 | 97500 |
| Select8 | 63000 | 64500 | 46500 |
| Select3Exchange | 57000 | 151500 | 55500 |

**3) Таблиця 4.2, N = 7500**

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 250500 | 235500 | 219000 |
| Select8 | 145500 | 141000 | 100500 |
| Select3Exchange | 123000 | 343500 | 121500 |

**4) Таблиця 4.3, N = 10000**

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 420000 | 429000 | 400500 |
| Select8 | 271500 | 247500 | 195000 |
| Select3Exchange | 210000 | 601500 | 214500 |

**5) Таблиця 4.4, N = 12500**

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 655500 | 696000 | 642000 |
| Select8 | 420000 | 390000 | 310500 |
| Select3Exchange | 331500 | 948000 | 336000 |

Загальні характеристики швидкодії різних алгоритмів

Наведені нижче значення показують усереднений за допомогою гармонічного середнього час, витрачений алгоритмами на сортування найскладніших випадків (окрім вектору):

**1) Таблиця 5.0**

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Відсортований | Невідсортований | Обернений |
| Select6 | 304913 | 386977 | 327463 |
| Select8 | 210684 | 278137 | 213692 |
| Select3Exchange | 169521 | 5912168 | 208027 |

Порівняльний аналіз алгоритмів

Загальнотеоретичне порівняння

**Складність алгоритмів**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Алгоритм | Випадки | | | | | |
| Найкращий | | Середній | | Найгірший | |
| Порівняння | Перестановки | Порівняння | Перестановки | Порівняння | Перестановки |
| Select6 | O(n2) | O(1) | O(n2) | O(n) | O(n2) | O(n) |
| Select8 | O(n2) | O(1) | O(n2) | O(n) | O(n2) | O(n) |
| Select3Exchange | O(n2) | O(1) | O(n2) | O(n) | O(n2) | O(n) |

Усі представлені алгоритми за теорією мають складність O(n2). Це також можна побачити, подивившись на практично отримані дані. Зокрема можна звернути увагу на таблиці 4.0-4.4 та їх графіки, які показаують таку залежність. Але порівнювати алгоритми за цією властивістю не варто, адже, як сказано раніше, вони усі по складності однакові для усіх початкових випадків.

**Швидкодія алгоритмів**

А ось за часом роботи алгоритми вже трохи відрізняються. Порівнявши їх продуктивність за швидкодією, можна виділити те, що алгоритм Select8 значно швидше Select6, але Select3Exchange показує ще кращі результати, проте тільки на відсортованих векторах. У випадку оберноної відсортованості цей алгоритм повільніший за Select8, а у випадку невідсортованості показує дуже погані, найгірші результати, що на практиці робить цей гібридний алгоритм найповільнішим з усіх.

Такий приріст швидкодії у Select8 у порівнянні з Select6 можна пояснити кількістю відсортованих зон у масиві, у Select6 вона одна та за один цикл зростає на один елемент, коли у Select8 їх дві й кожна ітерація зовнішнього циклу зменшуе кількість невідсортованих елементів на двійку. Звісно й порівнянь виконується більше, але загалом Select8 значно більш ефективний ніж Select6.

Такі ж слова про Select3Exchange сказати не можна. Кількість порівнянь у цього алгоритма у випадку відсортованого вектору менша, що дає йому перевагу у швидкодії, адже перестановки не виконуються. Проте у випадку невідсортованого вектору, цей алгоритм, замість запам’ятовування індексів максимальних та мінімальних елементів, їх переставляє, що не може не впливати дуже негативно на швидкодію. Велика кількість «зайвих» перестановок ставить Select3Exchange на останне місце за швидкодією на невідсортованих векторах, що ставить під сумніви практичну користь цього методу сортування взагалі.

У випадку оберненої відсортованості вектора Select3Exchange може наблизитися за швидкодією до Select8 за допомогою першого порівняння зовнішнього циклу, що значно зменшуе кількість перестановок. Але кількість порівнянь все ж таки більша, тому на векторах навіть у цьому випадку цей алгоритм є гіршим за Select8. Хоча навіть так через кількість відсортованих зон (2), Select3Exchange значно швидше сортує за Select6.

**Поведінка у різних випадках відсортованності**

Механізм роботи Select6 найстабільнішим з усіх алгоритмів, він показує майже однаково повільний результат в усіх випадках. Трошки іншу картину має Select8, який показує кращі результати у випадку невідсортованості та значно кращі у випадку оберненої відсортованості. Select3Exchange демонструє лише незначну відмінність між випадком відсортованості та оберненої відсортованості. Принаймні, незначну у порівнянні із просто космічною відмінністю між випадком невідсортованності, який витрачає значно більше часу й робить цей алгоритм деякою мірою непередбачуваним.

Дослідження І. Залежність часу роботи алгоритмів від довжини стовпчиків масива

У цьому випадку від алгоритмів можна очікувати приблизно таку ж саму різницю між один одним, як і в попередньому розділі. Це випливає із специфіки цього дослідження: дійсно, на цей раз потрібно відсортувати трьохвимірний масив, але самі алгоритми насправді працюють над одновимірним вектором сум стовпців, довжина якого є константою для усіх випадків. Але на цей раз відрізняється механізм перестановки елементів вектора, з ним також потрібно поміняти місцями стовпці переріза саме за допомогою копіювання їх вмісту (у цій курсовій роботі це робиться без використання допоміжного вектора, за допомогою обміну та допоміжної змінної), що може й буде впливати на швидкодію алгоритмів та додатково підкреслить недоліки деяких з них, у яких не все дуже оптимізовано з перестановками. Загалом, за теорією можна припустити, що із лінійним збільшенням кількості елементів у стовпці, буде й лінійно зростати кількість роботи, бо у всіх алгоритмів в усіх випадках виконується O(n) перестановок. В усіх, окрім вже відсортованих масивів, де можна очікувати невеликий приріст часу, зумовлений лінійною залежністю кількості елементів у стовпці та часу, потрібного на формування вектору суми.

Оскільки кількість елементів у стовпці зростає у геометричній прогресії, а кількість саме стовпців та перерізів залишається незмінною, можна спрогнозувати наступне:

1. Час роботи буде збільшуватись експоненціально.
2. Алгоритми будуть вести себе схожим чином як й у попередньому розділі (найповільніший залишиться найповільнішим, найшвидший найшвидшим тощо).
3. Select3Exchange буде значно повільнішим за усе інше для випадку невідсортованого масиву.

**Select6**

Цей алгоритм продовжує демонструвати стабільні загалом результати для усіх випадків відсортованості на усіх експериментах, окрім останніх, де можна побачити експоненціальний приріст часу для невідсортованого та обернено відсортованого випадку та ліній пририіст часу для відсортованого випадку.

**Select8**

Цей алгоритм веде себе аналогічно до Select6, демонструє підвищену швидкість роботи, зумолвену механізмом роботи алгоритму, в решті нічим цікавим не відзначається.

**Select3Exchange**

Цей алгоритм демонструє нам підвищену швидкодію у випадках відсортованості та оберненої відсортованості. На випадках з невідсортованим масивом часу витрачається дуже багато.

**Загальні спостереження**

Алгоритми поводять себе на практиці так, як було припущено із теорії. До того ж, час сортування вектору помножений на три приблизно дорівнює часу сортування 3д-масиву з одним елементом на стовпчик.

**Порівняння**

Проміжкові підсумкі схожі із загальнотеоретичними. Усі алгоритми в експериментах зберегли свої співвідношення між випадками та між собою. Але Select3Exchange відзначився високою повільністю, через що у задачах, схожих на це дослідження, використовувати його не варто, але можна взяти Select8.

**Аналіз**

* **Select6**: алгоритм показує стабільну поведінку, хоча вона трошки змінюється під останні експерименти, коли велика кількість елементів у стовпці змушує операції по перестановці стовпців стати надто важкими, що й виливається у приріст відносно відсортованого випадку.
* **Select8**: алгоритм показує свою звичайну поведінку на початкових експериментах, де кількість елементів у стовпцях невелика. Із зростанням кількості елементів росте й важкість перестановок та змушує випадок з невідсортованим масивом стати більш часозатратним. Під кінець з тієї ж самої причини «вартість» перстановок робить й випадок оберененої відсортованості більш часозатратним ніж випадок відсортованості. Загалом алгоритм залишається швидшим за Select6 завдяки тому, що за 1 ітерацію зменшує невідсортовану частину на 2 елементи.
* **Select3Exchange**: менша кількість порівнянь у випадку відсортованого масиву допомогає цьому алгоритму за швидкодією у цьому випадку обійти Select8 та Select6. Як і для інших алгоритмів зростаюча вартість перестановок виводить випадки відсортованості та оберненої відсортованості з рівноваги, де останній випадок стає більш часозатратним. Велика кількість перестановок, яку робить алгоритм, чинить у випадку невідсортованості жахливий вплив на швидкодію, який нічим не можна компенсувати. Це головний недостаток цього алгоритму та це дослідження додатково та дуже яскраво його підкреслює.

Дослідження ІI. Залежність часу роботи алгоритмів від форми перерізів масива

У цьому дослідженні кількість перерізів залишається надалі незмінною, але зменшується кількість стовпців, коли в одночас збільшується кількість їх елементів. Це негативно вплине на вартість перестановок, але необхідність у них буде зменшуватись. Можна припустити, що загалом із зменшенням N, незважаючи на збільшення M, час роботи буде зменшуватись.

**Select6**

Кількість витраченого часу, як і було припущено, поступово зменшується, але останній експеримент показує певну аномалію, де час трохи став більше. Також можна відзначити, що під кінець час, витрачений на обернено відсортовані масиви стає більшим за час, витрачений на вже відсортовані спочатку масиви.

**Select8**

Алгоритм у цьому дослідженні також показує кращі результати, ніж Select6. Витрачений на сортування вже відсортованих масивів зменшується, те ж саме можна сказати й про випадок невідсортованих масивів, хоча під кінець різниця у часі стає не дуже помітною. Час, витрачений на сортування обернено відсортованих масивів зменшується також, але останній експеримент демонструє аномальне підвищення часу.

**Select3Exchange**

Цей алгоритм знов показує свою перевагу тільки для випадків відсортованих масивів та обернено відсортованих. Обернено відсортовані масиви сортуються трошки швидже, ніж вже відсортовані, цю різницю краще видно на останніх експериментах. Для невідсортованих масивів результати такі ж погані, як і у попередніх дослідженнях, тут Select3Exchange значно повільніший за попередні алгоритми. Проте слід відзначити, що час, витрачений на сортування зменшується завжди, без аномалій у кінці, в цьому дослідженні цей алгоритм найбільш стабільний, така властивість може стати в нагоді у деяких задачах.

**Загальні спостереження**

Алгоритми поводять себе на практиці так, як було припущено із теорії, але у перших двох алгоритмах можна побачити певний сплеск у вимірюваннях.

**Порівняння**

Select8 залишається найшвидшим алгоритмом за середнім часом, але Select3Exchange залишається швидшим у випадках відсортованості та оберненої відсортованості, до того ж, повидиться більш передбачувано та стабільно, але його середній час значно гірший ніж у Select8 та Select6.

**Аналіз**

* **Select6**: як і за припущеннями, витрачений на сортування вже відсортованих масивів спадає. Із зростанням кількості елементів у стовпцях, перестановки стають більш часозатратними, що виводить з рівноваги час на сортування вже відсортованих та обернено відсортованих масивів. Під кінець стовпці стають настільки великими, що компенсувати це їхньою малою кількістю не можна, що й виливається у те, що випадки це кількість перестановок більше 0 (невідсортовані, обернено відсортовані) починають витрачати більше часу.
* **Select8**: кількість відсортованих зон знов допомагає алгоритму отримати кращий середній час. В решті поведінка схожа із Select6, час спадає, але наприкінці бере реванш та росте, бо перестановки стають тяжчими. Також можна звернути увагу на те, що кількість перестановок стає більшою на обернено відсортованому масиві, саме це є причиною сплеску часу на «оберненому» в останньому експерименті.
* **Select3Exchange:** алгоритм своєю великою кількістю перестановок знов ставить себе на останне місце по ефективності на випадках невідсортованості. Проте перша перевірка зовнішнього циклу (Sum[L] > Sum[R]) допомагає йому зберегти певну стабільність та лінійне спадання часу при спаданні N.

Дослідження ІII. Залежність часу роботи алгоритмів від кількості ключів у кожному перерізі масива при однаковій загальній кількості ключів у всьому масиві

У цьому дослідженні розмір масиву залишатиме себе однаковим в усіх експериментах, але кількість перерізів буде зменшуватись, а кількість стовпців буде зростати. Кількість елементів у стовпцях M буде незмінною та дорівнювати 100.

**Select6**

Із зменшенням кількості стовпців витрачений час зменшується, але не лінійно та не квадратично. Випадок невідсортованості більш «тяжкий» ніж випадок оберненої відсортованості. Сортування вже відсортованного масиву відбувається найшвидше.

**Select8**

Поведінка майже така ж сама як і у Select6, але сортування відбувається швидше та у останньому експерименті сортування оберненого випадку відбувається повільніше, ніж невідсортованого.

**Select3Exchange**

Сортування цим алгоритмом швидше у випадках відсортованості та оберненої відсортованості, але значно повільніше у решті випадків. Із зменшенням N (та збільшенням P) витрачений час зменшується за схожим паттерном.

**Загальні спостереження**

При рості P та зменшенні N алгоритми витрачають менше часу на сортування. При цьому можна спостерігати те, що час зменшується за якоюсь закономірністю, яка не є лінійною або квадратичною.

**Порівняння**

Select8 знову швидше за Select6. Select3Exchange продовжує бути найгіршим, окрім випадків відсортованості та оберненої відсортованості, де він швидше за Select8.

**Аналіз**

Проводити порівняльний аналіз тут не дуже доречно, адже алгоритми ведуть себе майже так само як і у попередніх дослідженнях. Швидкодія Select8, повільнодія Select3Exchange та причини таких ситуацій описані у попередніх параграфах, зокрема у розділі про вектори. А особливий паттерн зменшення часу при збільшенні P пов’язаний із специфікою дослідження, способу обходу та задачі. Справа у тому, що усі алгоритми мають однакову складність, квадратичну, але самі перерізи сортуються незалежно один від одного, сортуються насправді лише вектори сум. Тому, кількість перерізів та кількість стовпців впливають на час, але різною мірою, хоча за одним напрямком. Для усіх алгоритмів кількість перерізів впливає на час лінійно, а кількість стовпців квадратично. Саме через це при зменшенні N, але збільшенні P описаним у дослідженні чином, час падає, але за особливим, не лінійним та не квадратичним законом.

Висновки

* **Select6**

Оптимізований алгоритм сортування прямим вибором, класичний представник алгоритмів цього классу. Він показав тут не самі кращі результати, до того ж його складність не дозволяє використовувати його на великих масивах даних та надто довгих векторах.

* **Select8**

Ще більш оптимізований алгоритм сортування вибором. У цій курсовій він показав себе, на мою думку, краще за всіх. Його складність теж не дозволяє його використовувати у більшості практичних задач, але він більш швидкий ніж Select6 в усіх випадках, та обходить Select3Exchange у головному випадку на практиці.

* **Select3Exchange**

Із усіх присутніх тут алгоритмів, цей показав найгірші результати. Його гібридність дозволила йому отримати недоліки сортування обміном та вибором. Хоча він і швидше ніж Select8 у випадках оберненої відсортованості та простої відсортованості, різниця у швидкодії між цими випадками та випадком невідсортованості настількі космічна, що використовувати цей алгоритм просто неможливо. Навіть якщо припустити, що існує така задача, у якій потрібно мати справу з масивами, які здебільшого відсортовані або відсортовані навпаки, кращим варіантом буде первірити в яку сторону масив відсортовано та перевернути його або все-ж таки відсортувати, за потреби, чимось нормальним, або хоча б Select8.

* **Загалом**

Найкращим з практичної точки зору виявився Select8, але в усіх цих алгоритмах проблематична складність, яка несумісна із більшістю практичних задач. Використання таких алгоритмів вже призводило до поганих наслідків: нещодавно у ОС Windows знайшли баг, який унеможливлював нормальне користування робочим столом, тому що якийсь інженер не припустив те, що елементів буде так багато та використав алгоритм час роботи котрого росте квадратично від кількості вхідних даних. Такі алгоритми дуже легко спочатку зробити, пропустити поміж очей їх проблеми (не кожен буде сортувати в якості тесту великі масиви), а потім сумувати через втрачені обчислювальні цикли. Для сортування взагалі, зокрема і для описаних тут випадків значно краще підійде «покращена» версія алгоритмів сортування вибором: сортування кучею. Або взагалі сортування Хоара. У обох цих алгоритмів складність O(n log n) для усіх випадків (окрім коли усі елементи рівні, тоді O(n)), але сортування Хоара швидше, через що використовується у стандартних бібліотеках більшості мов програмування.

Література

1. Марченко О. І. Лекції з алгоритмів та струтур даних на YouTube – <https://www.youtube.com/channel/UCerCwtA87I0yuq3tTQCk7vw>
2. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн «Алгоритмы. Построение и анализ.» – 3-е вид. – СПб.: Диалектка, 2020.
3. Марченко О. І., Марченко Л.А. Программирование в среде Turbo

Pascal 7.0. – 9-е изд. – К.: Век+, Спб.: КОРОНА-Век, 2007.