Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Кафедра інформатики

КУРСОВА РОБОТА

Тема: “Порівняння методів сортування масивів”

з дисципліни «Програмування»

ПОЯСНЮВАЛЬНА ЗАПИСКА

Керівник \_\_\_\_\_\_\_\_\_\_\_ Руденко Д. О.

Студент гр. КІ-15-3 \_\_\_\_\_\_\_\_\_\_\_ Розенберг М. E.

Харків 2016

РЕФЕРАТ

Записка пояснювальна до курсової роботи: 25 с., 7 рис., 3 розділи, 1 додаток.

Мета роботи — розробка програми для порівняння таких методів сортування, як: метод злиття, швидке сортування та за допомогою купи.

Метод вирішення задачі — для вирішення поставленої задачі були проаналізовані основні алгоритми сортування масивів.

Розроблено програму, яка реалізує основні алгоритми сортування та дозволяє заміряти час виконання цих алгоритмів, що дозволяю зручно порівнювати швидкість виконання реалізованих методів.

Програму складено мовою C++ у середовищі програмування Visual Studio 2013.

СОРТУВАННЯ, С++, ШВИДКЕ СОРТУВАННЯ, СОРТУВАННЯ ЗЛИТТЯМ, СОРТУВАННЯ ЗА ДОПОМОГОЮ КУПИ, АЛГОРИТМ.

ЗМІСТ

[ВСТУП 4](#_Toc452221294)

[1. ТЕОРЕТИЧНА ЧАСТИНА 5](#_Toc452221295)

[1.1 Постановка задачі 5](#_Toc452221296)

[1.2 Сортування злиттям 5](#_Toc452221297)

[1.3 Швидке сортування 7](#_Toc452221298)

[1.4 Сортування за допомогою купи 9](#_Toc452221299)

[2. ПРОГРАМНА РЕАЛІЗАЦІЯ 13](#_Toc452221300)

[2.1 Описання структури програми 13](#_Toc452221301)

[2.2 Опис функції int main 13](#_Toc452221302)

[2.3 Опис функції int\* copyMainArray 13](#_Toc452221303)

[2.4 Опис функції void generateRandomMas 14](#_Toc452221304)

[2.5 Опис функції timer 14](#_Toc452221305)

[3. ІНСТРУКЦІЯ КОРИСТУВАЧА 15](#_Toc452221306)

[ВИСНОВКИ 18](#_Toc452221307)

[ПЕРЕЛІК ПОСИЛАНЬ 19](#_Toc452221308)

[Додаток А 20](#_Toc452221309)

[А.1 – файл ресукрсів main.cpp 20](#_Toc452221310)

[А.2 – файл ресукрсів mergeSort.cpp 22](#_Toc452221311)

[А.3 – файл ресукрсів quicksort.cpp 23](#_Toc452221312)

[А.4 – файл ресукрсів heapSort.cpp 24](#_Toc452221313)

[A.5 – заголовочний файл sorts.h 25](#_Toc452221314)

# ВСТУП

Алгоритм - набір інструкцій, що описують порядок дій виконавця для досягнення певного результату.

Часто в якості виконавця виступає комп'ютер, але поняття алгоритму необов'язково відноситься до комп'ютерних програм, так, наприклад, чітко описаний рецепт приготування страви також є алгоритмом, в такому випадку виконавцем є людина (а може бути і певний механізм, ткацький верстат, і ін.).

Поняття алгоритму належить до первісних, основним, базисним поняттям математики. Обчислювальні процеси алгоритмічного характеру (арифметичні дії над цілими числами, знаходження найбільшого загального дільника двох чисел і т. Д.) Відомі людству з глибокої давнини. Однак в явному вигляді поняття алгоритму сформувалося лише на початку XX століття.

Алгоритм сортування - це алгоритм для упорядкування елементів в списку. У разі, коли елемент списку має кілька полів, поле, що служить критерієм порядку, називається ключем сортування. На практиці в якості ключа часто виступає число, а в інших полях зберігаються будь-які дані, що не впливають на роботу алгоритму.

Алгоритми сортування оцінюються за швидкістю виконання та ефективності використання пам'яті:

Час - основний параметр, що характеризує швидкодію алгоритму. Називається також обчислювальною складністю. Для упорядкування важливі найгірша, середня і краща поведінка алгоритму.

Пам'ять - ряд алгоритмів вимагає виділення додаткової пам'яті під тимчасове зберігання даних. Як правило, ці алгоритми вимагають O (log n) пам'яті. При оцінці не враховується місце, яке займає вихідний масив і не залежать від вхідної послідовності витрати, наприклад, на зберігання коду програми (так як все це споживає O (1)).

# 1. ТЕОРЕТИЧНА ЧАСТИНА

1.1 Постановка задачі

Була поставлена задача розробити додаток, метою якого є порівняння методів сортування масивів. У якості цих методів були обрані такі алгоритмі:

* сортування злиттям;
* швидке сортування;
* сортування за допомогою купи.

Для реалізації можливості порівняння цих алгоритмів виникає необхідність в розробці засобів для виконання підрахунку витраченого часу на виконання впорядкування заданого масиву. Так як необхідно порівнювати декілька алгоритмів, то виникає необхідність реалізувати цей засіб таким чином, щоб виключити дуплікації коду та забезпечити зручне додавання нових методів для аналізу.

Також, для виконання порівнянь алгоритмів, необхідно буде для кожного методу надавати необхідні данні для сортування. Тому необхідно розробити можливість введення цих даних користувачем і збереження їх для подальшого використання у інших алгоритмах. Але на невеликому обсязі даних неможливо виконати перевірку швидкості виконання, тому цей додаток повинен надати можливість генерувати масиви з випадкових чисел заданої довжини, що суттєво збільшить швидкість роботи з цією програмою.

Розроблювана програма повинна мати простий користувальницький інтерфейс, який дозволив би виконувати усі необхідні завдання. Також програма повинна працювати у весь час, доки користувач не вибере необхідний пункт у меню для завершення програми.

1.2 Сортування злиттям

Сортування злиттям - алгоритм сортування, який впорядковує списки (або інші структури даних, доступ до елементів, яких можна отримувати тільки послідовно) в певному порядку. Ця сортування - хороший приклад використання принципу «розділяй і володарюй». Спочатку завдання розбивається на кілька підзадач меншого розміру. Потім ці завдання вирішуються за допомогою рекурсивного виклику або безпосередньо, якщо їх розмір досить малий. Нарешті, їх рішення комбінуються, і виходить рішення вихідної завдання.

Рішення даної задачі сортування можна зобразити у вигляді блок-схеми, представленої на рис. 1.1.

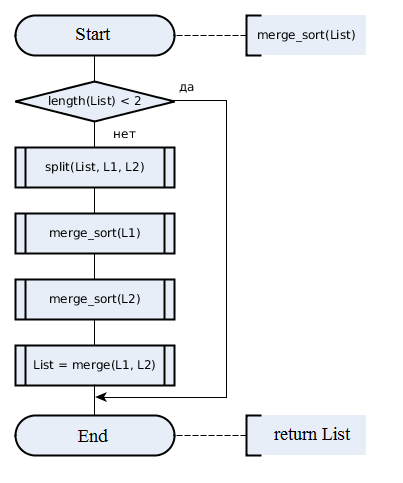


Рисунок 1.1 – «Блок-схема сортування злиттям»

Дана блок-схема має наступні кроки:

* Сортований масив розбивається на дві частини приблизно однакового розміру.
* Кожна з отриманих частин сортується окремо, наприклад - тим же самим алгоритмом. Рекурсивне розбиття на менші відбувається до тих пір, поки розмір масиву не досягне одиниці (будь-який масив довжини 1 можна вважати впорядкованим).
* Два упорядкованих масиву половинного розміру з'єднуються в один. Основну ідею злиття двох відсортованих масивів можна пояснити на наступному прикладі. Нехай ми маємо два вже відсортованих по зростанню підмасива. тоді злиття двох підмасивів у третій результуючий масив. На кожному кроці ми беремо менший з двох перших елементів підмасивів і записуємо його в результуючий масив. Лічильники номерів елементів результуючого масиву і підмасива, з якого був узятий елемент, збільшуємо на 1. Коли один з підмассивов закінчився, ми додаємо решту елементів другого підмасива в результуючий масив.

Алгоритм був винайдений Джоном фон Нейманом в 1945 році.

Час роботи алгоритму O (n \* log n) при відсутності деградації на невдалих випадках, які є болючим місцем швидкого сортування.

До переваг данного методу сортування можно віднести:

* Працює навіть на структурах даних послідовного доступу.
* Добре поєднується з підкачкою і кешуванням пам'яті.
* Непогано працює в паралельному варіанті: легко розбити завдання між процесорами порівну, але важко зробити так, щоб інші процесори взяли на себе роботу, в разі якщо один процесор затримається.
* Не має «важких» вхідних даних.

Але цей метод також має і свої недоліки:

* На «майже відсортованих» масивах працює так само довго, як на хаотичних.
* Вимагає додаткової пам'яті за розміром вихідного масиву.

## 1.3 Швидке сортування

Швидке сортування — алгоритм сортування, добре відомий, як алгоритм розроблений Чарльзом Гоаром, який не потребує додаткової пам'яті і виконує у середньому O(n\*log(n)) операцій. Однак, у найгіршому випадку робить O(n^2) порівнянь. Оскільки алгоритм використовує дуже прості цикли і операції, він працює швидше інших алгоритмів, що мають таку ж асимптотичну оцінку складності. Наприклад, зазвичай більш ніж удвічі швидший порівняно з сортуванням злиттям.

Ідея алгоритму полягає в переставлянні елементів масиву таким чином, щоб його можна було розділити на дві частини і кожний елемент з першої частини був не більший за будь-який елемент з другої. Впорядкування кожної з частин відбувається рекурсивно. Алгоритм швидкого сортування може бути реалізований як у масиві, так і в двозв'язному списку. Цей алгоритм може бути представлений у вигляді блок-схеми зображеної на рис. 1.2.

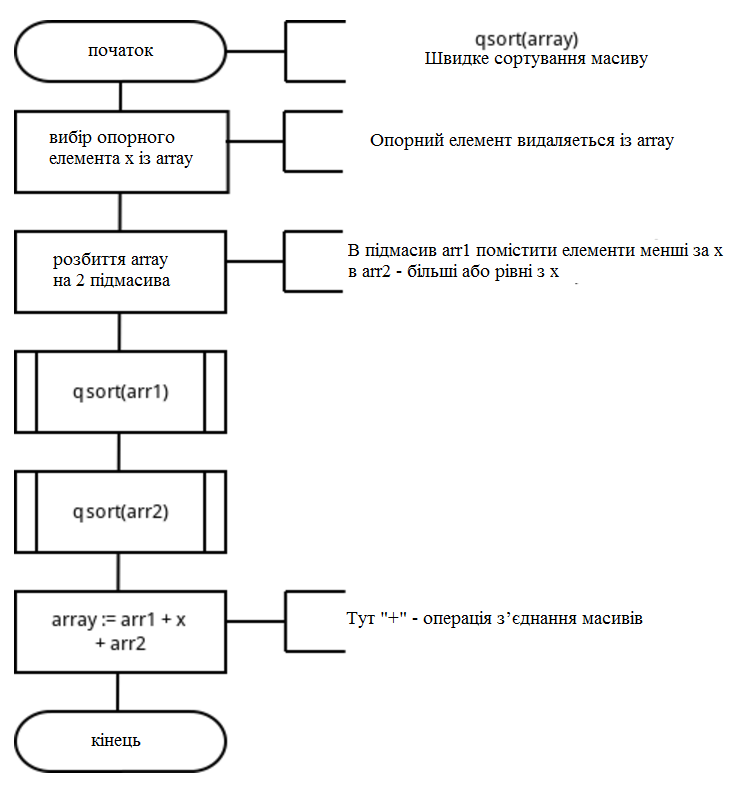


Рисунок 1.2 – «Блок-схема швидкого сортування»

Швидке сортування є алгоритмом на основі порівнянь, і не є стабільним.

Алгоритм швидкого сортування було розроблено Тоні Гоаром (C. A. R. Hoare) у 1962 під час роботи у маленькій британській компанії Elliott Brothers.

Час роботи алгоритму сортування залежить від збалансованості, що характеризує розбиття. Збалансованість, у свою чергу залежить від того, який елемент обрано як опорний (відносно якого елемента виконується розбиття). Якщо розбиття збалансоване, то асимптотично алгоритм працює так само швидко як і алгоритм сортування злиттям. У найгіршому випадку, асимптотична поведінка алгоритму настільки ж погана, як і в алгоритму сортування включенням.

Найгірша поведінка має місце у тому випадку, коли процедура, що виконує розбиття, породжує одну підзадачу з n-1 елементом, а другу — з 0 елементами. Нехай таке незбалансоване розбиття виникає при кожному рекурсивному виклику. Для самого розбиття потрібен час О(n). Тоді, рекурентне співвідношення для часу роботи, можна записати так:

T(n) = T(n-1)+T(0)+ θ (n) = T(n-1)+ θ(n).

Розв'язком такого співвідношення є T(n)=О(n^2).

В найкращому випадку процедура Partition ділить задачу на дві підзадачі, розмір кожної не перевищує n/2. Час роботи, описується нерівністю:

T(n) ≤ 2T(n/2) + θ(n).

Тоді:

T(n)= θ(n\*log n) — асимптотично найкращий час.

Математичне очікування часу роботи алгоритму на всіх можливих вхідних масивах є O(n\*log n), тобто середній випадок ближчий до найкращого.

В середньому алгоритм працює дуже швидко, але на практиці, не всі можливі вхідні масиви мають однакову імовірність. Тоді, шляхом додання рандомізації вдається отримати середній час роботи в будь-якому випадку.

## 1.4 Сортування за допомогою купи

Сортування купою або пірамідальне сортування — алгоритм сортування, працює в гіршому, в середньому і в кращому випадку (тобто гарантовано) за Θ(n log n) операцій при сортуванні n елементів. Кількість застосовуваної службової пам'яті не залежить від розміру масиву (тобто, O (1)).

Сортування пірамідою використовує бінарне сортувальне дерево. Бінарне сортувальне дерево — це структура даних, що є масивом, який можна розглядати як майже повне бінарне дерево. Кожен вузол цього дерева відповідає певному елементу масива. На всіх рівнях, крім, можливо останнього, дерево повністю заповнене (заповнений рівень — такий, що містить максимально можливу кількість вузлів). Останній рівень заповнюється послідовно зліва направо до тих пір, доки в масиві не закінчяться елементи.

Для масиву Array у корені дерева знаходиться елемент Array[1]. Далі дерево будується за наступним принципом: якщо якомусь вузлу відповідає індекс i, то індекс його батьківського вузла і/2, індекс лівого дочірнього вузла — 2і, а індекс правого дочірнього вузла 2і+1.

Розглядають два види бінарних куп: неспадні і незростаючі. В обох видах значення, що розташовані у вузлах купи, задовольняють властивості купи. Властивість незростаючої купи полягає в тому, що для кожного вузла крім кореневого виконується нерівність Array[i/2] ≥ Array[i]. Іншими словами, значення вузла не перевищує значення батьківського вузла. Таким чином найбільший елемент знаходиться в корені дерева.

Принцип побудови неспадної купи протилежний. Властивість неспадної купи полягає в тому, що кожен елемент крім кореневого є неменшим за свій батьківський елемент Array[i/2] ≤ Array[i].

Зручна структура даних для сортувального дерева — такий масив Array, що Array[1] — елемент в корені, а нащадки елемента Array[i] є Array[2i] і Array[2i+1].

Алгоритм сортування складатиметься з двох основних кроків:

* Вибудовуємо елементи масиву у вигляді сортувального дерева. Цей крок вимагає O(n) операцій.
* Будемо видаляти елементи з кореня по одному за раз і перебудовувати дерево. Тобто на першому кроці обмінюємо Array[1] і Array[n], перетворюємо Array[1], Array[2], … , Array[n-1] в сортувальне дерево. Потім переставляємо Array[1] і Array[n-1], перетворюємо Array[1], Array[2], … , Array[n-2] в сортувальне дерево. Процес продовжується до тих пір, поки в сортувальному дереві не залишиться один елемент. Тоді Array[1], Array[2], … , Array[n] — впорядкована послідовність. Цей крок вимагає O(n \ log n) операцій.

Отриманий алгоритм для сортування можна зобразити у вигляді блок-схеми, яка представлена на рис. 1.3.

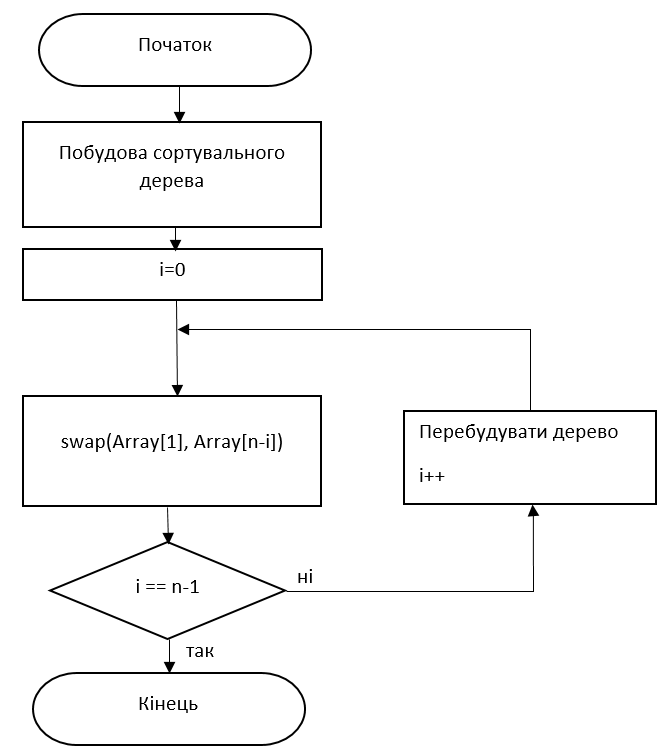


Рисунок 1.3 – Блок-схема сортування купою

Переваги алгоритму:

* час роботи в найгіршому випадку — O(n \log n);
* вимагає O(1) додаткової пам'яті.

Недоліки алгоритму:

* нестійкий — для забезпечення стійкості потрібно розширювати ключ;
* на майже відсортованих даних працює так само довго, як і на хаотичних даних;
* складний в реалізації;
* на одному кроці вибірку доводиться робити хаотично по всій довжині масиву — тому алгоритм погано поєднується з кешуванням і підкачкою пам'яті;
* методу потрібно «миттєвий» прямий доступ; не працює на зв'язкових списках та інших структурах пам'яті послідовного доступу.

# 2. ПРОГРАМНА РЕАЛІЗАЦІЯ

## 2.1 Описання структури програми

Отримана програма складеться з заголовного файлу «sorts.h», який містить прототипи усіх реалізованих методів сортування. Для кожного алгоритму сортування було зроблено окремий файл «назваМетоду.cpp». Це дозволило зробити код більш прозорим и зрозумілим одразу.

Основна логіка програми для виклику сортування та виконання додаткових функцій знаходиться у файлі main.cpp.

## 2.2 Опис функції int main

Це головна функція програми з якої починається її виконання. Напочатку цієї функції викликається метод setlocal, який дозволяє використовувати у користувальницькому меню програми російські літери. Далі відбувається створення змінної command, яка буде відповідати за номер пункту в меню, який обрав користувач.

Далі створено нескінченний цикл, який буде працювати доти, доки не буде обрано пункт, який дозволяє війти із програми. Усередині цього циклу відбувається вибір необхідної дії за допомогою оператора switxh, яку можна вибрати з меню програми.

## 2.3 Опис функції int\* copyMainArray

Дуже важливою функцією для програми є збереження масиву між запусками різних видів сортування. Але враховуюче те, що усі методи сортування змінюють існуючий масив, який передається їм на сортування, а не повертають новий відсортований масив, то виникаю необхідність у цій функції, яка займається тим, що копіює усі значення із головного масиву у тимчасовий, який потім використовується для сортування. У якості копіювання було обрану вбудовану функцію memcpy, яка на низькому рівні копію усі данні з одного масиву заданої довжини, до іншого. У якості результату ця функція повертає новий отриманий масив.

## 2.4 Опис функції void generateRandomMas

Ця функція була розроблена для надання користувачеві можливості тестувати алгоритми на великих об’ємах даних.

Для початку користувачеві пропонується увести розмір масиву, який необхідно буде заповнити випадковими числами. Після введення розміру масиву виконується цикл, який послідовно заповнює кожну комірку основного масиву програми випадковими числами, використовуючи вбудовану функцію rand.

## 2.5 Опис функції timer

Це одна із найважливіших функцій у системі. За допомогою неї відбувається замір часу виконання вибраної функції сортування.

В якості параметра ця функція приймає будь-яку функцію сортування. На початку її роботи перевіряється наявність у головному масиві елементів. У разі їх відсутності користувачу буде видано повідомлення про цю проблему. У іншому спочатку зберігається початкове значення часу. Потім відбувається копіювання масиву у інший тимчасовий за допомогою функції copyMainArray. І потім цей масив сортується функцією, яка передана у цей метод у якості параметра. Після виконання алгоритму знову зберігається значення часу, для того, щоб потім відобразити користувачу затрачений час на виконання сортування, який є різницею між кінцевим і початковим збереженим часом.

# 3. ІНСТРУКЦІЯ КОРИСТУВАЧА

Розроблена програма не потребую ніякої установки. Після запуску програми користувачеві відобразяться основні команди, які підтримує цей додаток. Це вікно має вигляд як на рис 3.1.

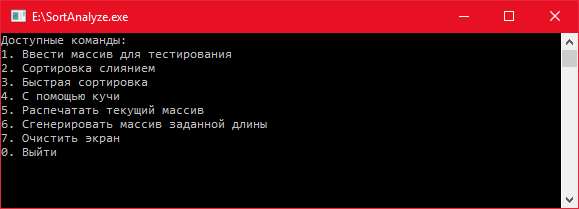


Рисунок 3.1 — Головні команди додатку

Для того, щоб розпочати аналізувати алгоритми для початку необхідно заповнити масив із даними, який потім буде використовуватись. Це можна зробити за допомогою двох функції: «Ввести массив для тестирования», «Сгенерировать массив заданной длины».

Коли користувач вибирає команду №1, то йому пропонується ввести майбутній розмір масиву, а потім через пробіл — усі числа масиву.

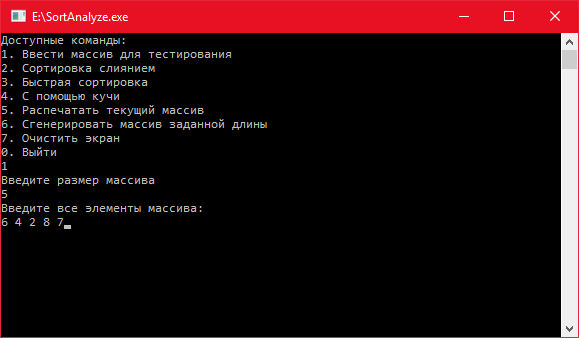


Рисунок 3.2 — Заповнення масиву користувачем

А у разі вибору команди №6 користувачеві пропонується лише ввести розмір масиву. А увесь масив заповниться автоматично випадковими числами.

Користувач завжди має можливість подивитися масив з яким він працює у конкретний момент. Для цього йому необхідно ввести команду №5. Тоді через пробіл надрукуються усі числа, які містяться у масиві. Якщо користувач ще не створив масив, то буде видано повідомлення про те, що у масиві немає елементів.

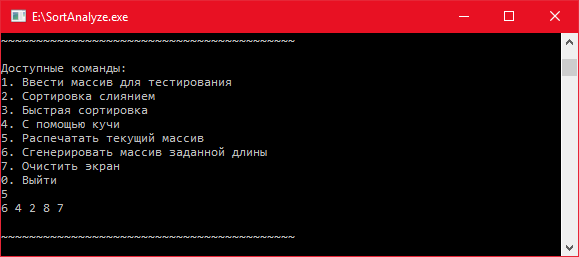


Рисунок 3.3 — Друкування поточного масиву

Після того, як користувач заповнив масив, то можна приступати до аналізу реалізованих алгоритмів. Для цього необхідно вибрати номер команди для потрібного методу сортування. Після цього програма почне сортування масиву. Коли ця операція буде закінчена, то час використаний для неї буде відображений на консолі. Час, який відображається вимірюється у мілісекундах.

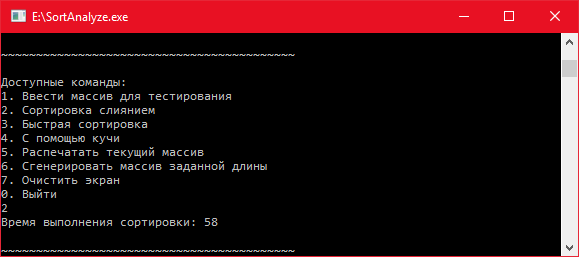


Рисунок 3.4 — Сортування масиву методом злиття

Так само можна виконувати і інші методи сортування. Для кожного наступного методу сортування буде використовуватися той самий масив, що дозволить порівняти ці алгоритму на однакових даних і точно визначити різницю у часі їх виконання.

У разі необхідності є можливість очистити екран від інформації яка вже була введена і більше не потрібна.

Для того, щоб вийти із програми можна або вибрати команду №0, або натиснути на хрестик консольного додатку.

# ВИСНОВКИ

У процесі виконання курсової роботи крім закріплення старих знань були придбані так само і нові.

Був розробленій додаток, який дозволяє виконувати сортування будь-якої послідовності чисел різними алгоритмами а також заміряти час їх виконання, що може допомогти у подальшому аналізі та дослідженні продуктивності алгоритмів сортування.

Також був розроблений універсальний засіб для заміру часу виконання вибраних алгоритмів. Це дозволить у майбутньому з легкістю розширювати кількість алгоритмів, які може аналізувати розроблений додаток.

Серед запропонованих алгоритмів сортування, завдяки розробленому додатку, був обраний найшвидший алгоритм. Хоча всі алгоритми мають однакову складність, все одно час їх виконання різниться. Найшвидше виконався алгоритм швидкого сортування. Він відсортував масив із 1000000 елементів за 330 мілісекунд. Але майже так само виконався і алгоритм сортування купою — за 340 мілісекунд. І довше за все виконувався алгоритм сортування злиттям — 5 секунд. Це пов’язано з тим, що для сортування злиттям необхідно створювати додатковий масив і виділяти під нього пам’ять. Через це з’являються додаткові затримки, які вкупі на великих даних зменшують швидкість виконання програми.

Всі ці набуті навички послужать міцною основою для подальшого отримання знань.

# ПЕРЕЛІК ПОСИЛАНЬ

1. В.М. Бондарев «Программирование на С++» [Текст] – Харьков: СМИТ, 2005г.
2. Подбельский В.В. Язык Си++: Учеб. пособие. [Текст] - М.: БИНОМ, 1995. - 400 с.
3. П. Путятiн, В.П. Степанов, В.П. Пчелiнов, Т.Г. Долженкова, О.О. Матат “Основи програмування мовою С++” [Текст] – Харьков: СМИТ, 2005г. – 320 с
4. Р. Седжвик «Алгоритмы на C++» [Текст] – Вильямс – 1056 с
5. Страуструп Б. Язык программирования C++, 3-е изд./ Пер. с англ. [Текст] -СПб.; М.: ”Невский Диалект” – “Издательство БИНОМ”,1999 г. – 991 с.

# Додаток А

Код програми

## А.1 – файл ресукрсів main.cpp

#include <iostream>

#include <locale>

#include "sorts.h"

int\* mainArray;

int length;

typedef void(\*SORTFUNCTION) (int\*, int);

bool isArrayInitiallized()

{

return length!=0;

}

void fillArray()

{

std::cout << "Введите размер массива" << std::endl;

std::cin >> length;

if (length <= 0)

{

std::cout << "Длина массива должна быть положительная" << std::endl;

length = 0;

return;

}

std::cout << "Введите все элементы массива:" << std::endl;

if (isArrayInitiallized())

{

delete[] mainArray;

}

mainArray = new int[length];

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

{

std::cin >> mainArray[i];

}

}

int\* copyMainArray()

{

int\* res = new int[length];

memcpy(res, mainArray, length\*sizeof(int));

return res;

}

void printArray(int\* array, int length)

{

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

{

std::cout << array[i] << " ";

}

std::cout << std::endl;

}

void generateRandomMas()

{

std::cout << "Введите размер массива" << std::endl;

std::cin >> length;

if (isArrayInitiallized())

{

delete[] mainArray;

}

mainArray = new int[length];

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

{

mainArray[i] = rand();

}

std::cout << "Массив сгенерирован" << std::endl;

}

void timer(SORTFUNCTION function)

{

if (!isArrayInitiallized())

{

std::cout << "Массив пустой" << std::endl;

return;

}

unsigned int start\_time = clock();

int\* tempArray = copyMainArray();

function(tempArray, length);

//printArray(tempArray, length);

delete[] tempArray;

unsigned int end\_time = clock();

unsigned int search\_time = end\_time - start\_time;

std::cout << "Время выполнения сортировки: " << search\_time << std::endl;

}

int main()

{

setlocale(LC\_ALL, "rus");

int command = -1;

length = 0;

while (command != 0)

{

std::cout << "Доступные команды:" << std::endl;

std::cout << "1. Ввести массив для тестирования" << std::endl;

std::cout << "2. Сортировка слиянием" << std::endl;

std::cout << "3. Быстрая сортировка" << std::endl;

std::cout << "4. С помощью кучи" << std::endl;

std::cout << "5. Распечатать текущий массив" << std::endl;

std::cout << "6. Сгенерировать массив заданной длины" << std::endl;

std::cout << "7. Очистить экран" << std::endl;

std::cout << "0. Выйти" << std::endl;

std::cin >> command;

switch (command)

{

case 1:

fillArray();

break;

case 2:

timer(mergeSort);

break;

case 3:

timer(quickSort);

break;

case 4:

timer(heapSort);

break;

case 5:

if (!isArrayInitiallized())

{

std::cout << "Массив пустой" << std::endl;

}

else

{

printArray(mainArray, length);

}

break;

case 6:

generateRandomMas();

break;

case 7:

system("cls");

break;

}

std::cout << "\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n" << std::endl;

}

return 0;

}

## А.2 – файл ресукрсів mergeSort.cpp

#include "sorts.h"

void merge(int\* const A, int const lengthA, int\* const B, int const lengthB, int\* C)

{

//Выполнить слияние массива A, содержащего nA элементов,

// и массива B, содержащего nB элементов.

// Результат записать в массив C.

int a = 0, b = 0; //Номера текущих элементов в массивах A и B

while (a + b < lengthA + lengthB) //Пока остались элементы в массивах

{

if ((b >= lengthB) || ((a < lengthA) && (A[a] <= B[b])))

{ //Копирую элемент из массива A

C[a + b] = A[a];

++a;

}

else { //Копирую элемент из массива B

C[a + b] = B[b];

++b;

}

}

}

void mergeSort(int\* array, int length)

{

if (length < 2) return; //Сортировка не нужна

if (length == 2) //Два элемента проще поменять местами,

{ // если нужно, чем делать слияние

if (array[0] > array[1])

{

int temp = array[0];

array[0] = array[1];

array[1] = temp;

}

return;

}

mergeSort(array, length / 2); //Сортируем первую половину

mergeSort(array + length / 2, length - length / 2); //Сортируем вторую половину

int\* tempArr = new int[length]; //Сюда запишем результат слияния

merge(array, length / 2, array + length / 2, length - length / 2, tempArr); //Слияние половин

//Копирование результата слияния в исходный массив:

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

{

array[i] = tempArr[i];

}

delete[] tempArr; //Удаляем временный буфер

}

## А.3 – файл ресукрсів quicksort.cpp

// нахождение опорного элемента

int partition(int\* a, int p, int r)

{

int x = \*(a + r);

int i = p - 1;

int j;

int tmp;

for (j = p; j < r; j++)

{

if (\*(a + j) <= x)

{

i++;

tmp = \*(a + i);

\*(a + i) = \*(a + j);

\*(a + j) = tmp;

}

}

tmp = \*(a + r);

\*(a + r) = \*(a + i + 1);

\*(a + i + 1) = tmp;

return i + 1;

}

void quickSort(int\* a, int p, int r)

{

int q;

if (p < r)

{

q = partition(a, p, r);

quickSort(a, p, q - 1);

quickSort(a, q + 1, r);

}

}

void quickSort(int\* a, int length)

{

quickSort(a, 0, length - 1);

}

## А.4 – файл ресукрсів heapSort.cpp

#include <iostream>

#include "sorts.h"

using namespace std;

// процедура просеивания следующего элемента

// До процедуры: a[k+1]...a[n] - пирамида

// После: a[k]...a[n] - пирамида

void downHeap(int a[], int k, int n) {

int new\_elem;

int child;

new\_elem = a[k];

while (k <= n / 2) { // пока у a[k] есть дети

child = 2 \* k;

// выбираем большего сына

if (child < n && a[child] < a[child + 1])

child++;

if (new\_elem >= a[child]) break;

// иначе

a[k] = a[child]; // переносим сына наверх

k = child;

}

a[k] = new\_elem;

}

void heapSort(int\* a, int length)

{

int temp;

// строим пирамиду

for (int i = length / 2 - 1; i >= 0; i--) downHeap(a, i, length - 1);

// теперь a[0]...a[length-1] пирамида

for (int i = length - 1; i > 0; i--) {

// меняем первый с последним

temp = a[i]; a[i] = a[0]; a[0] = temp;

// восстанавливаем пирамидальность a[0]...a[i-1]

downHeap(a, 0, i - 1);

}

}

## A.5 – заголовочний файл sorts.h

void mergeSort(int\* array, int length);

void quickSort(int\* a, int length);

void heapSort(int\* a, int length);