Міністерство освіти і науки України
Національний університет водного господарства та природокористування

Навчально-науковий інститут кібернетики, інформаційних технологій та інженерії
Кафедра обчислювальної техніки

04-04-296М

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних робіт з навчальної дисципліни
«Алгоритми та методи обчислень» (частина 1)
для здобувачів вищої освіти
першого (бакалаврського) рівня
за освітньо-професійною програмою
«Комп'ютерна інженерія»
спеціальності 123 «Комп'ютерна інженерія»
та освітньо-професійною програмою
«Інформаційна безпека»
спеціальності 125 «Кібербезпека та захист інформації»
денної та заочної форми навчання

Рекомендовано
науково-методичною радою
з якості ННІКІТІ
Протокол № 4 від 24.02.2025 р.

Рівне – 2025

Методичні вказівки до лабораторних робіт з навчальної дисципліни «Алгоритми та методи обчислень» (частина 1) для здобувачів вищої освіти першого (бакалаврського) рівня за освітньо-професійною програмою «Комп'ютерна інженерія» спеціальності 123 «Комп'ютерна інженерія» та освітньо-професійною програмою «Інформаційна безпека» спеціальності 125 «Кібербезпека та захист інформації» денної та заочної форми навчання. [Електронне видання] / Бойчура М. В., Сидор А. І. – Рівне : НУВГП, 2025. – 31 с.

Укладачі: Бойчура М. В., к.т.н., доцент кафедри обчислювальної техніки;
Сидор А. І., к.т.н., в.о. завідувача кафедри обчислювальної техніки.

Відповідальний за випуск: Сидор А. І., к.т.н., в.о. завідувача кафедри обчислювальної техніки.

Керівник групи забезпечення
спеціальності
123 «Комп'ютерна інженерія» Сидор А. І.

Керівник групи забезпечення
спеціальності
125 «Кібербезпека та захист інформації» Назарук В. Д.

ЗМІСТ

Вступ

Освітня компонента «Алгоритми та методи обчислень» відноситься до циклу дисциплін професійної підготовки бакалаврів низки спеціальностей галузі знань 12 «Інформаційні технології», зокрема 123 «Комп'ютерна інженерія» та 125 «Кібербезпека та захист інформації». Дані методичні вказівки розроблено у відповідності до змістовного модуля 1 (з назвою «Алгоритми») силабусу освітньої компоненти «Алгоритми та методи обчислень». Передбачається вивчення базових та спеціальних алгоритмів пошуку і сортування елементів у структурах даних. Важливим аспектом, при цьому, є набуття студентами компетентностей, які дозволятимуть їм не лише розробляти алгоритми розв'язання поставлених задач, але й визначати їх складність та обґрунтовувати правильність. Студенти удосконалять навички творчого мислення, розв'язуючи одні і ті ж задачі різними методами, при цьому, порівнюючи та аналізуючи отримані результати.

Метою першого модуля дисципліни «Алгоритми та методи обчислень» є надання студентам знань про суть алгоритмів, особливості типізації даних, складність алгоритмів та задач, реалізацію алгоритмів в мовах програмування високого рівня.

Завдання курсу включають ознайомлення студентів з: ключовими поняттями про алгоритми; основними типами та структурами даних на базі сучасних високорівневих мов програмування; базовими алгоритмами та підходами до пошуку даних і їх сортування.

В результаті вивчення дисципліни студент повинен набути компетентності для розробки алгоритмів та написання програм для обробки числової, текстової та більш складноструктурованої інформації. Очікується, що

здобувачами будуть розроблені бібліотеки алгоритмів пошуку та сортування у вигляді dll-фалів.

Студент повинен **знати**:

- основи алгоритмізації;
- теоретичні основи роботи алгоритмів пошуку та сортування;
- методи оцінки продуктивності та складності алгоритмів.

Студент повинен **вміти**:

- реалізовувати базові алгоритми пошуку і сортування;
- аналізувати ефективність алгоритмів;
- писати оптимізований, структурований і коректний код;
- тестувати програми на наборах даних, що включають крайні випадки;
- створювати бібліотеки (dll-файли), що включають реалізації алгоритмів;
- користуватися інструментами відлагодження для аналізу роботи алгоритмів.

Здобувачам дозволяється виконувати поставлені завдання за допомогою будь-яких сучасних мов програмування або проходити курси на освітніх платформах з подальшим перезарахуванням отриманих результатів у якості неформальної освіти. Проте варто попередньо узгодити всі деталі з викладачем.

У завданнях до лабораторних робіт часто містяться фрази виду: порахувати кількість простих/складних операцій порівняння/присвоєння. Слід відзначити, що складність операцій – це поняття відносне. Наприклад, операція множення потребує здійснення значно більшої кількості тактів процесора, ніж додавання; тоді як ділення – потребує ще більшої кількості тактів.

В межах даного курсу простими операціями вважатимуться ті, що в лівих та правих операндах містять лише дії додавання та віднімання. Складними операціями, відповідно, вважатимемо все інше. Це, наприклад, операції, в яких задіяні дії множення, ділення, розіменування чи доступу до елемента масиву або ж стандартні математичні функції тощо. Варто пам'ятати, що операція індексації масиву, наприклад, у C++ здійснюється із застосуванням вказівників, і, взагалі кажучи, із перевантаженим додаванням числа:

```
a[i] == *(a + i);
```

по цій причині доступ до елемента масиву вважатимемо складною операцією.

До операцій порівняння відносяться наступні:

```
>  <  >=  <=  ==  !=
```

До операцій присвоєння відноситимемо оператор

```
=
```

Приклади

```
a[i] = 5;//складна операція присвоєння
x = 4;//проста операція присвоєння
x = a[11];//складна операція присвоєння
x = y = z = 11;//три прості операції присвоєння
if (x > 5)//просте порівняння
if (x >= 2 && x > a[j])//одне просте порівняння
    та одне складне порівняння
if (x)//проста операція порівняння, оскільки (x)
    неявно заміняється на (x == true)
```

Лабораторна робота №1
Найпростіші алгоритми пошуку

Мета

- Навчити студентів реалізовувати алгоритми пошуку в масиві, оцінювати оптимальність коду та аналізувати складність використаних операцій.
- Набуття навичок для зниження складності алгоритмів з точки зору кількості використання ресурсоємких операцій.

Завдання

1. Метод прямого пошуку елемента:
    - реалізувати алгоритм прямого пошуку елемента в масиві;
    - вивести результат у форматі «позиція_числа – шукане_число»;
    - зафіксувати кількість простих і складних операцій порівняння та присвоєння у коді.
2. Метод прямого пошуку всіх входжень:
    - реалізувати алгоритм прямого пошуку всіх входжень елемента в масиві;
    - вивести результати у форматі «позиція_числа – шукане_число»;
    - зафіксувати кількість простих і складних операцій порівняння та присвоєння у коді.
3. Метод бінарного пошуку:
    - реалізувати алгоритм бінарного пошуку елемента у впорядкованому масиві;
    - вивести результат у форматі «позиція_числа – шукане_число»;
    - зафіксувати кількість простих і складних операцій порівняння та присвоєння у коді.

4. Створити власну бібліотеку dll, куди добавити розроблені алгоритми.

Вимоги до розв'язків завдань

- Код повинен бути структурованим та оптимальним.
- Алгоритми повинні бути протестовані хоча б на 5 суттєво різних (з точки зору тестування) наборах даних.
- Оформлення виведення результатів повинне відповідати форматам, описаним у завданнях.
- Забороняється використовувати вбудовані методи пошуку; всі алгоритми повинні бути реалізовані вручну.

Критерії оцінювання

1. Правильність реалізації алгоритмів (40%).
2. Оптимальність коду та відповідність вимогам (30%).
3. Аналіз кількості простих та складних операцій порівняння і присвоєння (20%).
4. Якість тестування програми (10%).

Контрольні запитання

1. Що таке алгоритм пошуку?
2. У чому полягає різниця між прямим і бінарним пошуком?
3. Які фактори впливають на оптимальність алгоритму?
4. Як оцінюється складність алгоритму?
5. Які типи пошуку можна використовувати для невпорядкованих масивів?
6. Що таке складність O(n)?
7. Чому бінарний пошук не працює для невпорядкованих масивів?
8. Як реалізувати перевірку оптимальності алгоритму?
9. Які серед наведених вище алгоритмів є найефективнішими для великих масивів?
10. Що таке прості та складні операції?

11. Як визначити кількість операцій у коді?
12. Які способи покращення алгоритму Ви знаєте?
13. Як впливає структура даних на вибір алгоритму?
14. Що станеться, якщо бінарний пошук застосувати до невпорядкованого масиву?
15. Як перевірити коректність роботи алгоритму?
16. Як впливає розмір масиву на швидкість роботи алгоритму?
17. Яка різниця між ітеративною та рекурсивною реалізацією?
18. Що таке найгірший випадок у випадку дослідження ефективності алгоритму?
19. Що таке найкращий випадок у випадку дослідження ефективності алгоритму?
20. Як оптимізувати пошук у відсортованому масиві?
21. Як визначити, чи є число в масиві, використовуючи бінарний пошук?
22. Як правильно тестувати алгоритм?
23. Як оптимізувати кількість операцій у коді?
24. Як пов’язати ефективність алгоритму з його складністю?
25. Чому важливо враховувати обсяг пам’яті під час реалізації алгоритму?
26. Чому важливо порівнювати ефективність алгоритмів на однакових наборах даних?
27. Як програмно перевірити чи є масив впорядкованим?
28. Як використовуються індекси під час пошуку?
29. Чому важливо тестувати алгоритм на крайніх випадках?
30. Як реалізується пошук у багатовимірних масивах?
31. Як уникнути перевантаження пам’яті при великих масивах?
32. Як здійснити ефективний пошук у масиві, що зберігає об’єкти?

Лабораторна робота №2
Алгоритми пошуку підпослідовності у послідовності

Мета

- Навчити студентів працювати з алгоритмами пошуку підпослідовностей у послідовності, зокрема з алгоритмом Кнута-Морріса-Пратта.
- Набуття навичок використання інструментів відлагодження (Debugging).
- Удосконалення навичок для зниження складності алгоритмів з точки зору кількості використання ресурсоємких операцій.

Завдання

1. Прямий метод пошуку підпослідовності:
    - реалізувати алгоритм прямого пошуку підпослідовності у послідовності;
    - вивести індекс першого елемента бази, де має місце входження підпослідовності у послідовність;
    - порахувати кількість простих і складних операцій порівняння та присвоєння у коді.
2. Алгоритм Кнута-Морріса-Пратта:
    - реалізувати алгоритм Кнута-Морріса-Пратта для пошуку підпослідовності у послідовності;
    - вивести:
        - індекс першого елемента бази, де має місце входження підпослідовності у послідовність;
        - значення вектора префіксів;
    - пояснити результати на основі виведених даних;
    - програмно порахувати кількість простих і складних операцій окремо для функції префіксів і основного циклу.
3. Добавити розроблені алгоритми у власну бібліотеку dll.

4. Порівняння алгоритмів:
    - виконати тестування обох алгоритмів на декількох наборах даних;
    - порівняти кількість операцій порівняння та присвоєння в обох реалізаціях.

Вимоги до розв'язків завдань

- Код має бути структурованим та оптимальним.
- Заборонено використовувати вбудовані функції пошуку; всі алгоритми повинні бути реалізовані вручну.
- Результати тестування оформлюються у форматі:
    - «індекс: значення» для пошуку;
    - кількість операцій порівняння та присвоєння.
- Алгоритми повинні бути протестовані хоча б на 5 наборах даних. Можна, наприклад, використати наступні бази та образи:

```
int a[] = { 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3,
     4, 5 };
int b[] = { 1, 2, 3, 1, 2, 3, 1, 2, 3, 4, 5 };
```

```
int a[] = { 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2,
     2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2,
     2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2,
     2, 2, 2, 2, 2, 2, 2, 2 };
int b[] = { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 };
```

```
int a[] = { 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2,
     2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2,
     2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2,
     2, 2, 2, 2, 2, 2, 2, 2 };
int b[] = { 2, 2, 3, 2, 2, 2, 3, 2, 2, 2, 2 };
```

```
int a[] = { 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2,
     2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2,
     2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2,
     2, 2, 2, 2, 2, 2, 2, 2 };
int b[] = { 2, 2, 3, 2, 4, 4, 3, 2, 2, 4, 4 };
```

- Зважаючи на складність алгоритму Кнута-Морріса-Пратта, в якості виключення, у даній лабораторній роботі

наводиться відповідний фрагмент коду програми (потрібно детально розібратися із цим кодом; може знадобитись інструмент Debug; для цього, наприклад, в MS Visual Studio передбачені команди, що виконуються за допомогою клавіш F5, F9, F10):

```
void Prefix()
{
    p[0] = -1;
    k = -1;
    for (i = 1; i < m; ++i)
    {
        while (k > -1 && b[k + 1] != b[i])
        {
            k = p[k];
        }
        if (b[k + 1] == b[i])
        {
            ++k;
        }
        p[i] = k;
    }
}

int main()
{
    Prefix();
    k = -1;
    for (i = 0; i < n; ++i)
    {
        while (k > -1 && b[k + 1] != a[i])
        {
            cout << k;
            k = p[k];
        }
        if (b[k + 1] == a[i])
        {
            ++k;
        }
        if (k == m - 1)
        {
            cout << i - m + 1 << endl;
            k = p[k];
        }
    }
}
```

Критерії оцінювання

1. Правильність реалізації алгоритмів (30%).
2. Оптимальність коду та відповідність вимогам (30%).
3. Аналіз кількості простих та складних операцій порівняння і присвоєння (20%).
4. Якість тестування програми (10%).
5. Вміння користуватись інструментами відлагодження (10%).

Контрольні запитання

1. Що таке підпослідовність?
2. Як працює алгоритм Кнута-Морріса-Пратта?
3. Що таке префікс-функція?
4. У чому полягає перевага алгоритму Кнута-Морріса-Пратта у порівнянні з прямим пошуком?
5. Які фактори впливають на ефективність пошуку підпослідовності у послідовності?
6. Як проаналізувати складність алгоритму пошуку підпослідовності у послідовності?
7. Чим корисні інструменти відлагодження (Debugging)?
8. З якою метою варто рахувати кількість операцій?
9. Які проблеми можуть виникнути, якщо у випадку алгоритму Кнута-Морріса-Пратта вхідні дані не відсортовані?
10. Що таке часова складність алгоритму?
11. Як впливають повторювані числа у масиві на ефективність наведених вище алгоритмів?
12. У яких випадках доцільно використовувати прямий пошук?
13. Як поводиться наведені вище алгоритми з порожніми масивами?
14. Як забезпечити коректність роботи наведених вище алгоритмів при роботі з символами Unicode?

15. Як впливає довжина підпослідовності на ефективність пошуку у випадку наведених вище алгоритмів?

16. Яким чином можна порівняти ефективність алгоритмів пошуку підпослідовностей?

17. Які реальні задачі вирішує алгоритм Кнута-Морріса-Пратта?

18. Як модифікувати наведені вище алгоритми, щоб вони здійснювали пошук підпослідовності лише у фрагменті бази (від індексу A до індексу B)?

19. Як модифікувати наведені вище алгоритми так, щоб результат пошуку був коректний у випадку співпадіння для хоча би `m - 1` символів (а не всіх `m`)?

20. Чи працюватиме алгоритм Кнута-Морріса-Пратта без розрахунку масиву префіксів?

Лабораторна робота №3
Алгоритм Бауера-Мура пошуку елемента в масиві

Мета

- Здобути навички розуміння коду, написаного на незнайомій мові програмування.
- Розвинути навички абстрагування.
- Навчитися працювати із досить ефективним методом Бауера-Мура для пошуку елемента в масиві.
- Розвинути навички використання інструментів відлагодження (Debugging).
- Удосконалити навички для зниження складності алгоритмів з точки зору кількості використання ресурсоємких операцій.

Завдання

1. Перекласти алгоритм Бауера-Мура (наведений нижче) з мови Pascal на будь-яку іншу сучасну мову програмування (Python, C++, C#, Java тощо; цікаво, що Pascal багато в чому схожий на Python)
2. Зважаючи на те, що у Pascal індексація масивів починається з 1, а в багатьох сучасних мовах – з 0, реалізувати алгоритм відповідно до правил індексації обраної мови програмування.
3. Добавити розроблений алгоритм у власну бібліотеку dll.
4. Протестувати алгоритм на різних наборах даних:
    - масиви із короткими та довгими підпослідовностями;
    - масиви, які містять шукану підпослідовність та масиви, які не містять шукану підпослідовність;
    - особливі випадки, наприклад, порожні масиви чи масиви з декількома однаковими елементами.

5. Визначити кількість простих та складних операцій порівняння та присвоєння у коді.

6. Порівняти ефективність алгоритму Бауера-Мура з методом прямого пошуку у масиві.

7. Оптимізувати об’єм оперативної пам’яті, який потребує алгоритм.

Один із варіантів реалізації алгоритму Бауера-Мура

```
a: array[1..n] of byte;//base
b: array[1..m] of byte;//image
d: array[0..255] of byte;
begin
     for x := 0 to 255 do
          d[x] := m;
     for i := 1 to m - 1 do
          d[b[i]] := m - i;
     j := m;
     k := m;
     while (j >= 1) and (k <= n) do
     begin
          j := m;
          i := k;
          while (j >= 1) and (a[i] == b[j]) do
          begin
               i := i - 1;
               j := j - 1;
          end;
          k := k + d[a[i]];
     end;
     if j < 1 then
          writeln('')
     else
          writeln('not found');
end.
```

Вимоги до програмної реалізації

- Код повинен бути структурованим та оптимальним.
- Реалізація алгоритму має враховувати основні принципи обраної мови програмування.
- Забороняється використовувати вбудовані методи пошуку; всі алгоритми повинні бути реалізовані вручну.

- Результати роботи програми повинні мати зрозумілий формат виведення.
- Алгоритм повинен бути протестованим хоча б на 5 наборах даних.

Критерії оцінювання

1. Правильність реалізації алгоритму (30%).
2. Оптимальність коду та відповідність вимогам (30%).
3. Аналіз кількості простих та складних операцій порівняння і присвоєння (20%).
4. Якість тестування програми (10%).
5. Вміння користуватись інструментами відлагодження (10%).

Контрольні запитання

1. Що таке алгоритм Бауера-Мура і як він працює?
2. В чому особливість нумерації елементів масиву у мові Pascal?
3. Які переваги має алгоритм Бауера-Мура у порівнянні з прямим пошуком?
4. Що таке таблиця зсувів у алгоритмі Бауера-Мура?
5. Як формується таблиця зсувів у Pascal?
6. Що відбувається, якщо елемент образу не знайдено в базі?
7. Як поводиться алгоритм Бауера-Мура у випадку порожнього масиву?
8. Яка різниця у швидкості між алгоритмом Бауера-Мура та методом прямого пошуку?
9. Як впливає довжина образу на ефективність роботи алгоритму Бауера-Мура?
10. Чи можливий вихід за межі масиву в алгоритмі Бауера-Мура? Якщо так, то як це попередити?
11. Чому таблиця зсувів ініціалізується значенням `m`?

12. Як перевірити оптимальність реалізації алгоритму Бауера-Мура?

13. Як відреагує алгоритм Бауера-Мура, якщо масив бази складатиметься з усіх однакових елементів?

14. Як можна використати інструменти налагодження для вивчення роботи алгоритму Бауера-Мура?

15. Чи можна реалізувати алгоритм Бауера-Мура рекурсивно?

16. Як впливає розмір таблиці зсувів на пам’ять і час виконання програми?

17. Чи можна адаптувати алгоритм Бауера-Мура для двовимірних масивів?

18. Як алгоритм Бауера-Мура може бути використаний у практичних задачах?

19. Як використати наведені вище алгоритми для текстового пошуку?

20. Як залежить швидкодія від розташування образу в базі у випадку обох наведених вище алгоритмів?

21. Як впливає довжина бази на швидкодію у випадку наведених вище алгоритмів?

22. Як наведені вище алгоритми обробляють повторювані елементи образу?

23. Як впливає кількість зсувів на час виконання?

24. Як поведуть наведені вище алгоритми, якщо образ є довшим за базу?

Лабораторна робота №4
Алгоритми сортування методами прямого обміну

Мета

- Ознайомитися із найбільш відомими алгоритмами сортування методами прямого обміну («камінець», «бульбашка», шейкерне сортування).
- Розвинути навички використання інструментів відлагодження (Debugging).
- Удосконалити навички для зниження складності алгоритмів з точки зору кількості використання ресурсоємких операцій.

Завдання

1. Реалізувати алгоритм сортування методом «бульбашки» для довільної послідовності чисел.
2. Реалізувати алгоритм сортування методом «камінця» для довільної послідовності чисел.
3. Реалізувати алгоритм шейкерного сортування для довільної послідовності чисел.
4. Добавити розроблені алгоритми у власну бібліотеку dll.
5. Для усіх трьох алгоритмів програмно порахувати:
    - кількість простих та складних операцій порівняння;
    - кількість простих та складних операцій присвоєння;
    - загальну кількість виконаних операцій.
6. Порівняти ефективність алгоритмів та вивести результат у вигляді таблиці (див. приклад табл. 1).

Таблиця 1

Приклад таблиці з порівнянням кількості простих та складних операцій деяких відомих алгоритмів обміну

Алгоритм	«Бульбашкове» сортування	«Камінцеве» сортування	«Шейкерне» сортування
Складні порівняння	84	96	61
Прості порівняння	99	113	71
Складні присвоєння	81	81	81
Прості присвоєння	136	151	105
Загальна кількість операцій	400	441	318

Вимоги до програмної реалізації

- Код має бути структурованим та оптимальним.
- Забороняється використання вбудованих функцій сортування; всі алгоритми повинні бути реалізовані вручну.
- Результати роботи програми повинні мати зрозумілий формат виведення.
- Необхідно забезпечити коректність роботи програми для масивів із різними наборами даних (короткі, довгі, від'ємні значення, повторення елементів тощо).
- Алгоритми повинні бути протестовані хоча б на 5 наборах даних.

Критерії оцінювання

1. Правильність реалізації алгоритмів сортування (30%).
2. Оптимальність коду та відповідність вимогам (30%).
3. Порівняльний аналіз ефективності алгоритмів у вигляді таблиці (20%).
4. Якість тестування програми (10%).

5. Вміння користуватись інструментами відлагодження (10%).

Контрольні запитання

1. Що таке сортування методом прямого обміну?
2. Як працює алгоритм методу «бульбашки»?
3. У чому відмінність між «бульбашковим» і «камінцевим» сортуванням?
4. Що таке шейкерне сортування? Чим воно відрізняється від «бульбашкового»?
5. У якій мірі різниться ефективність сортування для довгих та коротких масивів?
6. Які випадки є найгіршими для роботи алгоритмів сортування методом прямого обміну?
7. Чи можна адаптувати ці алгоритми для сортування списків замість масивів?
8. Яка перевага шейкерного сортування у порівнянні з «бульбашковим»?
9. Наскільки різниться ефективність кожного з алгоритмів за умови, що послідовність впорядкована в оберненому порядку?
10. Як реагують усі 3 методи сортування у випадку, коли вихідний масив містить лише однакові елементи?
11. У чому різниця між внутрішнім і зовнішнім циклами в «бульбашковому» сортуванні?
12. Що відбудеться, якщо поміняти місцями «метод бульбашки» та «метод камінця» у шейкерному сортуванні?
13. Який має вплив розмір масиву на час виконання програми при сортуванні методами прямого обміну?
14. Чому метод прямого обміну не часто використовується для великих масивів?
15. Як реалізувати шейкерне сортування у двовимірних масивах?

16. Які реальні задачі вирішуються методом прямого обміну?

17. Яким чином використовуються алгоритми прямого обміну у випадку структурованих даних?

18. Що відбувається у «бульбашковому» сортуванні, якщо масив уже впорядкований?

19. Як оцінити ефективність роботи шейкерного сортування?

20. Чи обов’язковим є здійснювати обмін сусідніх елементів з однаковими значеннями у випадку «камінцевого» сортування?

Лабораторна робота №5
Сортування вибором, включенням та швидким включенням (алгоритмом Шелла)

Мета

- Набути навички розробки алгоритмів сортування прямим включенням, прямим вибором та алгоритмом Шелла.
- Розвинути навички оптимізації алгоритмів сортування.
- Розвинути навички використання інструментів відлагодження (Debugging).
- Удосконалити навички для зниження складності алгоритмів з точки зору кількості використання ресурсоємких операцій.

Завдання

1. Реалізувати алгоритм прямого включення для сортування масивів:
    - порахувати кількість простих та складних операцій порівняння;
    - порахувати кількість простих та складних операцій присвоєння.
2. Реалізувати алгоритм прямого вибору для сортування масивів:
    - визначити кількість простих та складних операцій порівняння;
    - визначити кількість простих та складних операцій присвоєння.
3. Реалізувати алгоритм Шелла для сортування масивів:
    - порахувати кількість простих та складних операцій порівняння;
    - порахувати кількість простих та складних операцій присвоєння.

4. Добавити розроблені алгоритми у власну бібліотеку dll.

5. Звести результати аналізу ефективності кожного алгоритму у вигляд таблиці (див. приклад табл. 2).

Таблиця 2

Приклад таблиці з порівнянням кількості простих і складних операцій алгоритмів прямого включення, прямого вибору та Шелла

Алгоритм	Пряме включення	Прямий вибір	Алгоритм Шелла
Складні порівняння	45	55	30
Прості порівняння	50	60	40
Складні присвоєння	40	30	20
Прості присвоєння	60	70	50
Загальна кількість операцій	195	215	140

Вимоги до програмної реалізації

- Код має бути структурованим та оптимальним.
- Забороняється використання вбудованих функцій сортування; всі алгоритми повинні бути реалізовані вручну.
- Програма повинна коректно працювати з масивами різних розмірів та типів (від'ємні, додатні, нульові значення, повторення елементів тощо).
- Необхідно забезпечити зручний та інформативний вивід результатів роботи програми.
- Алгоритми повинні бути протестовані хоча б на 5 наборах даних.

Критерії оцінювання

1. Коректність реалізації алгоритмів (30%).
2. Правильність розрахунку кількості операцій (20%).
3. Оформлення результатів у вигляді таблиці (20%).
4. Оптимізація одного з алгоритмів з поясненням змін (20%).
5. Структурованість та зрозумілість коду (10%).

Контрольні запитання

1. Що таке сортування методом прямого включення?
2. Як працює алгоритм прямого вибору?
3. У чому полягає принцип сортування алгоритмом Шелла?
4. Які основні відмінності між прямим вибором та прямим включенням?
5. Як обирається інкремент в алгоритмі Шелла?
6. Як ступінь початкової впорядкованості елементів вхідного масиву впливає на ефективність кожного з алгоритмів?
7. Яким є найгірший випадок для роботи алгоритмів прямого включення та вибору?
8. Які переваги алгоритму Шелла у порівнянні з іншими методами сортування?
9. Чому сортування вибором є порівняно неефективним для великих масивів?
10. У яких випадках краще використовувати сортування методом прямого включення?
11. Як забезпечити коректність усіх трьох алгоритмів для випадку масивів, що містять однакові елементи?
12. Яка роль індексів у роботі алгоритмів прямого вибору та включення?
13. Чи можна адаптувати алгоритм Шелла для сортування списків?

14. Як впливає розмір масиву на ефективність сортування у всіх трьох випадках наведених вище алгоритмів?

15. Як впливає кількість елементів на кількість порівнянь у методі вибору?

16. Чому алгоритм Шелла краще працює з великими масивами?

17. Як вибрати оптимальні параметри для алгоритму Шелла?

18. Як реалізувати сортування методом прямого включення для двовимірних масивів?

19. Які особливості обробки порожніх масивів та масивів з єдиним елементом у випадку наведених вище алгоритмів?

20. Як забезпечити універсальність сортування для різних типів даних?

21. Як впливає рівень початкової впорядкованості даних на швидкодію алгоритму прямого вибору?

22. Які практичні задачі вирішуються наведеними вище алгоритмами?

23. Як впливає наявність однакових за значенням чисел у масиві на ефективність роботи алгоритмів?

24. Що потрібно змінити наведених вище алгоритмах для реалізації сортування в зворотному порядку?

25. Як оцінити продуктивність алгоритму для різних типів даних?

Лабораторна робота №6
Сортування швидким обміном (алгоритмом Quick Sort)

Мета

- Засвоїти принцип роботи алгоритму швидкого сортування (Quick Sort).
- Засвоїти підхід до розділення масиву на підмасиви за принципом опорного елемента.
- Розвинути навички використання інструментів відлагодження (Debugging).
- Удосконалити навички для зниження складності алгоритмів з точки зору кількості використання ресурсоємких операцій.

Завдання

1. Реалізувати алгоритм Quick Sort для сортування масиву.
2. Визначити кількість простих та складних операцій порівняння.
3. Визначити кількість простих та складних операцій присвоєння.
4. Добавити розроблений алгоритм у власну бібліотеку dll.
5. Провести тестування алгоритму на масивах різного розміру (10, 100, 1000 елементів) та визначити середню кількість операцій.
6. Оптимізувати об’єм оперативної пам’яті, який потребує алгоритм.
7. Звести результати у таблицю для кожного тестового масиву вигляд таблиці (див. приклад табл. 3).

Таблиця 3

Приклад таблиці з порівнянням кількості простих і складних операцій алгоритму Quick Sort

Назва критерію	Експеримент №1	Експеримент №2	Експеримент №3
Розмір масиву	10 елементів	100 елементів	1000 елементів
Об’єм пам’яті	40	80	120
Складні порівняння	15	450	5000
Прості порівняння	20	500	6000
Складні присвоєння	10	300	4000
Прості присвоєння	25	550	7000
Загальна кількість операцій	70	1800	22000

Вимоги до програмної реалізації

- Код повинен бути структурованим та оптимальним.
- Забороняється використовувати вбудовані функції сортування; всі алгоритми повинні бути реалізовані вручну.
- Необхідно забезпечити коректність роботи програми для масивів із різними наборами даних (короткі, довгі, від’ємні значення, повторення елементів тощо).
- Алгоритми повинні бути протестовані хоча б на 5 наборах даних.
- Результати роботи програми повинні мати зрозумілий формат виведення.

Критерії оцінювання

1. Правильність реалізації алгоритмів сортування (30%).

2. Оптимальність коду та відповідність вимогам (30%).

3. Порівняльний аналіз ефективності алгоритмів у вигляді таблиці (20%).

4. Якість тестування програми (10%).

5. Вміння користуватись інструментами відлагодження (10%).

Контрольні запитання

1. Що таке алгоритм Quick Sort?
2. Які основні етапи роботи алгоритму швидкого сортування?
3. Що таке «опорний елемент» у Quick Sort?
4. Як виконується розділення масиву на підмасиви у Quick Sort?
5. Чому Quick Sort є одним із найшвидших алгоритмів сортування?
6. Яким є найгірший випадок вхідних даних у випадку алгоритму Quick Sort?
7. Як вибір опорного елемента впливає на ефективність алгоритму?
8. Які типи масивів найкраще підходять для стандартного Quick Sort?
9. Як алгоритм Quick Sort можна адаптувати для роботи зі списками?
10. Чому Quick Sort погано підходить для сортування дуже малих масивів?
11. Які є хороші альтернативи алгоритму Quick Sort для сортування невеликих наборів даних?
12. Як впливає розмір масиву на швидкодію алгоритму Quick Sort?
13. Чому Quick Sort краще працює з випадково впорядкованими даними?
14. У яких реальних задачах застосовується Quick Sort?

15. Як модифікувати алгоритм Quick Sort для сортування двовимірних масивів?

16. Чому Quick Sort часто використовується в якості стандартного алгоритму сортування у багатьох мовах програмування?

17. Що відбувається у випадку алгоритму Quick Sort, якщо масив початково вже є впорядкованим?

18. Чи можливо реалізувати Quick Sort без рекурсії?

19. Який рівень ефективності від випадкового вибору опорного елементу у випадку алгоритму Quick Sort?

20. Яка оцінка складності об’єму використовуваної пам’яті для Quick Sort?

21. Які типи задач найгірше підходять для Quick Sort?

22. Чому важливо враховувати тип даних при виборі алгоритму сортування?

23. Як реалізувати алгоритм Quick Sort для сортування рядків?

Рекомендована література

1. Мелешко Є. В., Якименко М. С., Поліщук Л. І. Алгоритми та структури даних : навч. посіб. Кропивницький : Лисенко В.Ф., 2019. 156 с.
2. Ткачук В. М. Алгоритми та структура даних : навч. посіб. Івано-Франківськ : ПНУ ім. В. Стефаника, 2016. 286 с.
3. Новотарський М. А. Алгоритми та методи обчислень : навч. посіб. Київ : НТУУ «КПІ ім. Ігоря Сікорського», 2019. 407 с.
4. Креневич А. П. Алгоритми і структури даних : підручник. Київ : ВПЦ «Київський Університет», 2021. 200 с.
5. Кормен Т. Алгоритми доступно / пер. з англ. К. Яценко. Київ : К.І.С., 2021. 194 с.
6. Грудзинський Ю. Є. Алгоритми та структури даних: навч. посіб. Київ : НТУУ «КПІ ім. Ігоря Сікорського», 2022. 2015 с.
7. Гребенюк С. М., Кудін О. В., Лісняк А. О., Столярова А. В. Алгоритми та структури даних : навч. посіб. Запоріжжя : Запорізький національний університет, 2022. 128 с.
8. Шаховська Н. Б., Голощук Р. О. Алгоритми та структури даних. Львів : Магнолія 2006, 2024. 216 с.
9. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 4th ed. Cambridge : The MIT Press, 2022. 1312 p.
10. Wengrow J. A common-sense guide to data structures and algorithms: level up your core programming skills. 2nd ed. Dallas : Pragmatic Bookshelf, 2020. 508 p.
11. Кублій Л. І. Алгоритми та структури даних. Основи алгоритмізації : підручник. Київ : НТУУ «КПІ ім. Ігоря Сікорського», 2022. 528 с.