

Міністерство освіти і науки України
Департамент науки і освіти Харківської обласної державної адміністрації
Харківське територіальне відділення МАН України

Відділення: комп'ютерні науки
Секція: комп'ютерні системи та мережі

РОЗРОБКА АЛГОРИТМУ ПОШУКУ В КОМП'ЮТЕРНИХ СИСТЕМАХ

Роботу виконав:
Попович Ярослав Васильович,
учень 9 класу Харківського
Навчально-виховного комплексу
№45 «Академічна гімназія»
Харківської міської ради
Харківської області

Науковий керівник:
Руккас Кирило Маркович,
професор кафедри теоретичної та
прикладної інформатики
механіко-математичного
факультету Харківського
національного університету
імені В.Н.Каразіна, доктор
технічних наук, доцент

Харків – 2019

Розробка алгоритму пошуку найкоротшого шляху в комп'ютерних мережах

Автор роботи:Попович Ярослав Васильович;

Харківське територіальне відділення МАН України;

Харківський навчально-виховний комплекс №45 «Академічна гімназія»

Харківської міської ради Харківської області; 9 клас; м. Харків;

Науковий керівник: Руккас Кирило Маркович, професор кафедри теоретичної та прикладної інформатики механіко-математичного факультету Харківського національного університету імені В.Н.Каразіна, доктор технічних наук, доцент.

1)Вступ.

З'явлення персональних комп'ютерів зажадало нового підходу до організації системи обробки даних, до створення нових інформаційних технологій. Використання комп'ютерних мереж потребувало створення великої кількості програм для обробки різної інформації.

Процес створення комп'ютерної програми для вирішення будь-якої практичної задачі складається з декількох етапів. Я розгляну етапи розробки алгоритму і тестування на прикладі програми, яка зможе знайти оптимальний варіант розподілу дітей по групам у математичному таборі. Основною метою цієї роботи є порівняння різноманітних алгоритмів для вирішення цієї задачі.

Дане дослідження може бути застосоване при розробці комп'ютерних програм, вхідні данні якої є одномірний упорядкований масив, та треба знайти елемент чи під масив за особливими критеріями.

2)Результати.

В результаті проведеної роботи були розглянуті два відомих алгоритму пошуку, та розроблені два алгоритму. Була створена програма по розподілу дітей в групи для деяких занять по критеріям. Було проаналізовано кілька алгоритмів пошуку на прикладі цієї задачі.

ЗМІСТ

Вступ.....	4
1. Характеристика алгоритмів пошуку в одномірних масивах.....	6
1.1. Лінійний пошук.....	6
1.2. Алгоритм двійкового пошуку (бінарний пошук)	7
2 Власна розробка алгоритмів пошуку	10
2.1. Власний алгоритм пошуку А	12
2.2. Власний алгоритм пошуку Б	16
3. Порівняльний аналіз алгоритмів	20
3.1 Розробка тестів та тестування алгоритмів.....	20
3.2. Порівняльна характеристика часу роботи алгоритмів.....	21
Висновки.....	25
Список використаних джерел	26

ВСТУП

З'явлення персональних комп'ютерів зажадало нового підходу до організації системи обробки даних, до створення нових інформаційних технологій. Виникла необхідність у переході від використання окремих комп'ютерів до об'єднання їх у комп'ютерну систему.

Ці комп'ютери, які об'єднані до єдиної системи, дозволили отримати доступ до необмежених інформаційних ресурсів, що, в свою чергу, потребувало створення великої кількості програм для обробки різної інформації.

Процес створення комп'ютерної програми для вирішення будь-якої практичної задачі складається з декількох етапів: формалізація і створення технічного завдання на вихідну задачу, розробка алгоритму вирішення задачі, написання, тестування, наладка і документування програми, отримання розв'язку вихідної задачі шляхом виконання програми.

Розглянемо етапи розробки алгоритму і тестування на прикладі наступного завдання: треба розробити програму, яка зможе знайти оптимальний варіант розподілу дітей по групам у математичному таборі на підставі віку, полу, рівня математичних знань, міста проживання, номеру школи, класу та спортивних вподобань. Для заповнення даних програми використовуються інтернет-анкети учасників, які зібрані із різноманітних міст України та різноманітних навчальних закладів. Ці дані зібрані в особову картку учасника, і підлягають до обробки, програмою. В результаті можна ставити завдання для програми, використовуючи один чи кілька перерахованих вище критеріїв.

Основна мета цієї роботи — порівняння різноманітних алгоритмів для вирішення цієї задачі.

Для реалізації поставленого завдання найкраще підходять алгоритми пошуку в одномірному масиві, так як інформація про кожного учня є елементом масиву.

Алгоритми пошуку мають широке застосування в індексації(пошуку) ресурсів комп'ютерних систем та мереж. Наприклад пошук файлів із різними типами, адреси, імена різних сервісів, процесів у комп'ютерних системах та мережах.

На прикладі розробленої програми розглянемо та проаналізуємо вже відомі алгоритми пошуку, та порівняємо їх із двома новими(винайденими) алгоритмами.

Для виконання цієї задачі необхідно:

1. Охарактеризувати алгоритми пошуку в одномірних масивах
2. Розглянути вже відомі алгоритми пошуку
3. Розглянути методи розробки алгоритмів
4. Створити нові алгоритми
5. Створити тести
6. Протестувати кожен алгоритм
7. Порівняти роботу алгоритмів пошуку
8. Знайти оптимальний варіант для вирішення цієї задачі

Дане дослідження може бути застосоване при розробці комп'ютерних програм, вхідні данні якої — одномірний упорядкований масив, та треба знайти елемент чи під масив за особливими критеріями.

РОЗДІЛ 1

ХАРАКТЕРИСТИКА АЛГОРИТМІВ ПОШУКУ В ОДНОМІРНИХ МАСИВАХ

1.1. Лінійний пошук

Лінійний, послідовний пошук — це алгоритм знаходження заданого значення довільної функції на деякому відрізку. Алгоритм лінійного пошуку є найпростішим (у реалізації) з усіх інших алгоритмів пошуку. [2]

В нього є декілька переваг та недоліків. Розглянемо їх детальніше.

Почнемо з того, що алгоритм послідовного пошуку не накладає ніяких обмежень на функцію, на відмінно, наприклад, від бінарного пошуку. Тобто його можна застосувати до від'ємних чисел (у нашій задачі ми цього не будемо робити).

Але в цього алгоритму є й недоліки. Наприклад, через те, що цей алгоритм перевіряє кожний елемент в масиві (зазвичай починаючи з лівого боку, з найменших чисел), в нього велика асимптотика: $O(n)$. З цього випливає, що алгоритм довго працює.

Алгоритм лінійного пошуку зазвичай використовують тільки при роботі з маленькими масивами, чи коли масив не відсортований.

Якщо треба знайти під масив (однакових елементів) у масиві, можна скористатися двома різними реалізаціями:

- Якщо у масиві багато різних елементів, і мало однакових, можна запустити алгоритм лінійного пошуку з лівого краю. Коли знайдемо потрібне число, запам'ятаймо його позицію, і коли знайдемо не рівне йому число запам'ятаймо позицію на 1 ділянку лівіше, а потім закінчимо йти.

- Якщо у масиві багато однакових елементів, можна запустити послідовний пошук з лівого краю, а потім ще раз з правого. Наприклад, якщо в нас є такий масив: 1, 2, 2, 2, 3, і ми хочемо знайти де починається входження двійок, набагато швидше буде запустити алгоритм спочатку з лівого краю, а потім з правого, ніж спочатку з лівого, а потім продовжити йти далі. [1]

Розглянемо цей алгоритм на прикладі задачі про математичний табір.

Протестувавши код із 100 тестів за допомогою компілятора GPU (C++ AMP), можна побачити, що час роботи алгоритму при різних тестах склав від 1.53 секунди до 1.764 секунди, що занадто багато для виконання цієї операції. При збільшенні навантаження на систему до 10000 тестів, у кожному з яких до 100000 чисел у масиві, час виконання роботи зіставить 180.423 секунди.

В цілому, цей алгоритм добре підходить для виконання простих завдань (коли мало елементів у масиві), коли потрібно працювати з від'ємними числами. А також, якщо виникає потреба дізнатись чи є у масиві той чи інший елемент. Також для роботи цього алгоритму масив не обов'язково має бути відсортованим, на відмінно за інші алгоритми.

Для виконання задачі про табір використання цього алгоритму не є оптимальним через великий час виконання роботи.

1.2. Алгоритм двійкового пошуку (бінарний пошук)

Також для пошуку найкоротшого шляху в одномірному масиві використовують двійковий (бінарний) пошук. Цей алгоритм є найпоширенішим алгоритмом пошуку.

Двійковий пошук — це алгоритм знаходження заданого значення упорядкованому масиві, який полягає у порівнянні серединного елемента масиву з шуканим значенням, і повторенням алгоритму для тієї або іншої половини, залежно від результату порівняння. [2]

Зрозуміти як працює цей алгоритм дуже легко. Програма бере середній елемент (за знаходженням) у масиві і дивиться, якщо цей елемент дорівнює тому, що шукається, то треба вийти з програми (запам'ятовуючи місце знаходження, наявність, и т.д.), інакше програма дивиться, якщо цей (середній) елемент менший за той, що шукається, то запускаються ті самі дії праворуч від середнього елемента, інакше ліворуч.

На кожній ітерації алгоритму масив ділиться на 2 частини. І так робиться доки не залишиться 1 елемент. Тобто, якщо масив ділиться на дві частини n раз, то в масиві знаходиться 2^n елементів. Із цього робиться висновок, що для знаходження потрібного елемента потрібно $\log n$ ітерацій. З цього робиться висновок, що асимптотика цього алгоритму є $O(\log_2 n)$.

Також слід відмітити, що цей алгоритм буде працювати тільки у відсортованому масиві, на відмінно від лінійного пошуку, що є великим недоліком.

Зазвичай пишуть одну з двох версій алгоритму двійкового пошуку: ітеративна та рекурсивна.

Розглянемо обидві версії алгоритму на прикладі задачі про табір.

В ітеративній версії бінарного пошуку використовується цикл, який змінює значення змінних L і R (вони відповідають за обмеження у під масиві, який ми розглядаємо), доки різниця між ними не стане ≤ 1 .

Рекурсивна версія бінарного пошуку відрізняється тим, що програма на кожній ітерації алгоритму запускає сама себе, але з іншими параметрами.

Але це не єдина відмінність у реалізації цих алгоритмів. Рекурсивна версія записується швидше і легше, ніж ітеративна, але використовує більше пам'яті тому, що на кожній ітерації алгоритму програма створює нові змінні для запуску самої себе.

Ще в цього алгоритму є перевага, що його можна використовувати, навіть, коли в масиві є елементи з від'ємним значенням.

Протестуємо цей алгоритм на компіляторі GPU (C++ AMP), і ітеративне рішення на 100 тестах працювало від 1.09 секунди до 1.221 секунди. На 10000 тестах ітеративне рішення працювало приблизно 110 секунд.

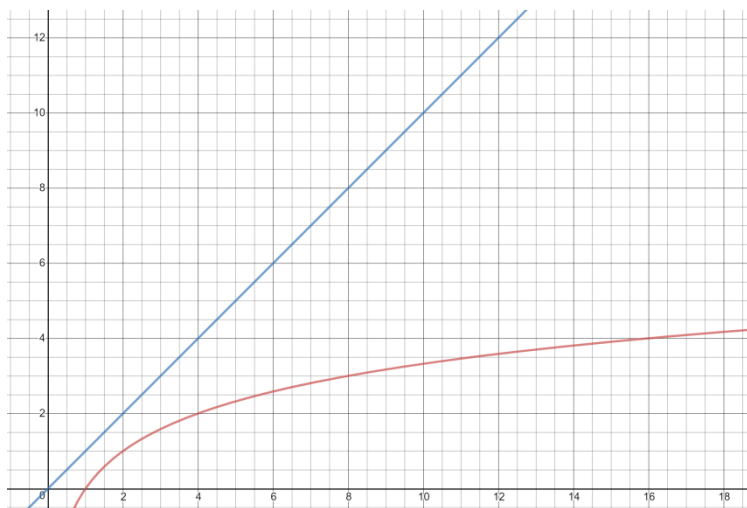
Рекурсивне рішення майже так само. На 100 тестах воно працювало від 1.047 секунд до 1.202 секунд. А на 10000 тестах рекурсивне рішення працювало приблизно 111 секунд.

Можемо зробити висновок, що ітеративне рішення працює так само, як і рекурсивне.

Також можна помітити, що на малих тестах лінійний пошук працює приблизно як і бінарний, але зі збільшенням тестів, збільшується і час роботи. Це можна легко пояснити. Якщо накреслити функцію $y = \log_2 x$ та функцію $y = x$, можна побачити, що при маленьких значеннях змінної X , Y у обох функціях не сильно відрізняються друг від друга.

Мал. 1.1

Функція $y = \log x$ і функція $y = x$



Для рішення задачі про табір цей алгоритм повністю підходить тому, що він працює досить швидко, і його асимптотика не від чого не залежить (на відміну від наступних алгоритмів).

РОЗДІЛ 2

ВЛАСНА РОЗРОБКА АЛГОРИТМІВ ПОШУКУ

Алгоритм — це набір інструкцій, які описують порядок дій виконавця для досягнення якогось результату. [1]

На даний момент розроблений ряд методів, які дозволяють отримувати ефективні алгоритми для вирішення великих класів завдань. До найвідоміших (та, мабуть, найважливіших) відносяться методи декомпозиції, динамічного програмування, «жадібні» методи пошуку з повертанням і локального пошуку.

Метод декомпозиції (або метод розбиття, або метод «розділяй і володарюй») є одним із найпоширеніших методів проектування алгоритмів. Він передбачає таке розбиття завдання розміру n на більш малі під завдання, так що в основі вирішення цих більш малих під завдань можна легко отримати рішення вихідної задачі. Як правило, алгоритми, які розроблені цим методом відрізняються легкістю розробки і високою ефективністю.

Динамічним програмуванням у найбільш загальній формі називають процес покрокового вирішення задачі, коли на кожному кроці вибирається одне рішення з купи допустимих на цьому кроці рішень, при тому таке, яке оптимізує задану цільову функцію. Цей процес покрокового рішення задачі полягає в створенні таблиць рішень всіх під задач, які доводиться вирішувати для отримання загального рішення.

«Жадібний» алгоритм — алгоритм, що полягає в прийнятті локально оптимальних рішень на кожному етапі, допускаючи, що кінцеве рішення також виявиться оптимальним.

Метод пошуку з повертанням (або метод повного перебору) застосовується у завданнях пошуку оптимального рішення, коли не вдається застосувати не один з описаних вище алгоритмів. Зазвичай це останній засіб для рішення задач, коли розглядаються усі можливі варіанти, серед яких і шукають оптимальне рішення.

Можна помітити, що алгоритм лінійного пошуку — це метод повного перебору, тому що алгоритм проходить по кожному елементу масиву та чекає доки не знайде потрібний.

Алгоритм має мати наступний набір властивостей:

Результативність — алгоритм повинен працювати швидко. Але як зрозуміти швидко працює алгоритм чи ні? Алгоритми розподіляються за швидкістю по такому принципу:

$O(1)$ — константний час (виявлення парності числа, представленого у двійковому форматі)

$O(\log_2 n)$ — логарифмічний час (зазвичай за такий час працюють алгоритми пошуку)

$O(n)$ — лінійний час (пошук суми чисел на відріжку в масиві)

$O(n \cdot \log_2 n)$ — лінійно-логарифмічний час (Quick Sort або Merge Sort)

$O(n^2)$ — квадратний час (сортування бульбашкою)

$O(n^3)$ — кубічний час (множення матриць)

$O(n!)$ — факторіальний час (задача про комівояжера) [2]

Для вирішення задачі про табір усі алгоритми, що будуть працювати довше за логарифмічний час не підходять, тому метод повного перебору можна викреслити.

Правильність — результат повинен бути завжди вірним. Тому «жадібний» алгоритм також не підходить.

Дискретність — алгоритм розбитий на окремі кроки (команди). Через це метод динамічного програмування не підходить.

Залишився лише метод декомпозиції. Алгоритм, який буде працювати швидше бінарного пошуку, та який буде працювати за постійний час у нас навряд вийде зробити, тому я пропоную зробити алгоритм, асимптотика якого буде залежати від відповіді. Наприклад якщо число, яке ми шукаємо порівняно з іншими буде дуже мале, то час виконання роботи буде зменшений, а якщо великий, то час виконання роботи алгоритму буде збільшений.

2.1 Власний алгоритм пошуку А (алгоритм ділення на три)

Гарний алгоритм пошуку має бути ефективним, тому методо повного перебору та «жадібний» алгоритм не підходять. Алгоритм бінарного пошуку написаний методом розбиття, тому що алгоритм кожен раз ділив масив на дві частини і працював тільки з однією, а також відповідь з'являлась тільки коли алгоритм розіб'є масив на досить багато частин і вирішить найменшу. Цей алгоритм працює досить добре, тому ми можемо спробувати створити алгоритм, який в деяких випадках буде працювати швидше за вищеописаний алгоритм.

Алгоритм бінарного пошуку розбивав масив на дві частини на кожній ітерації. Якщо на кожній ітерації розбивати масив на число частин більше ніж два(наприклад три) і перевіряти кожен частину, то цей алгоритм буде працювати довше, тому, що для розбиття масиву на три частини треба більше часу, ніж для розбиття на дві, тому не треба розбивати масив більше, ніж на дві частини.

Алгоритм А має багато схожого на бінарний пошук, але й відрізняється від нього. Алгоритм А так само, як і бінарний пошук може працювати з від'ємними числами, і цей алгоритм вміє працювати тільки з відсортованими масивами, так само, як і алгоритм ділення пополам.

Алгоритм А має рекурсивну реалізацію і працює дуже просто.

Він ділить масив у відношенні 1:2 (зліва), і дивиться якщо елемент на межі двох частин менше, ніж той, що шукається, то цей алгоритм ще раз запускається для більшої з двох частин, якщо елемент на межі буде більшим за той, що шукається, то алгоритм запускається для меншої з двох частин. Із цього робиться, що при малому шуканому числі цей алгоритм буде працювати набагато швидше за бінарний пошук, а при збільшенні значення шуканого числа порівняно з усіма числами масиву, буде витрачено більше часу.

Порахуймо найменшу (коли значення шуканого числа дорівнює значенню першого елементу масиву) асимптотику для алгоритму А, і найгіршу асимптотику (якщо значення шуканого числа дорівнює значенню останнього елементу масиву). У найвдалішому випадку програма буде кожен раз потрапляти до меншої частини,

і тоді асимптотика буде дорівнювати $O(\log_3 n)$, і на відмінно від бінарного пошуку, де логарифм був другої степені, тут він третьої степені, що набагато краще. З'ясуємо чому буде саме така асимптотика. Якщо алгоритм зробить N операцій, то в масиві буде 3^n елементів, отже при асимптотиці $\log_3 n$, буде зроблено $\log_3(3^n) = n$ операцій. Це можна порахувати математично.

$$n \cdot \left(\frac{1}{3^k}\right) = 1$$

$$n = 3^k$$

$$k = \log_3 n$$

Тепер порахуймо найбільшу (коли значення шуканого числа дорівнює значенню останнього елементу масиву) асимптотику алгоритму А. На кожній ітерації алгоритму програма буде відкидати одну третю частину елементів від тих, що залишилися. У найгіршому випадку програма k раз відкидає одну третю частину масиву (тобто залишається дві треті довжини масиву) із n елементів. А наприкінці цих дій має залишитись одне число. Зіставимо та вирішимо наступне рівняння:

$$n \cdot \left(\frac{2^k}{3^k}\right) = 1$$

$$n = \frac{3^k}{2^k}$$

$$n = \left(\frac{3}{2}\right)^k$$

$$n = 1.5^k$$

$$k = \log_{1.5} n$$

Якщо потрібно знайти під масив однакових елементів у масиві, можна це зробити за допомогою алгоритму А двома способами.

Перший є тривіальним. Просто потрібно запустити такий пошук окремо для лівої ділянки і окремо для правої.

Другий буде працювати швидше (знову не у всіх випадках). А відрізняється від першого він тим, що нам потрібно запустити пошук правої ділянки не від початку масиву, а від, вже знайденої, лівої ділянки під масиву. Він буде

працювати швидше за перший спосіб тому, що кількість елементів, яку треба перевірити буде меншою, ніж у першому випадку, та й елемент на розділі двох ділянок буде ближчий до лівого краю, ніж у першому випадку.

Цей алгоритм, хоч і має свої переваги, але він дуже нестабільний (у різних випадках працює за різну асимптотику), тому використовувати його для рішення моєї задачі буде не найоптимальнішою ідеєю, але цей алгоритм може бути використаний, коли відомо, що потрібний вам елемент буде малим.

Чим більше разів цей алгоритм відкине більшу частину, тим менше часу він буде працювати. На кожній ітерації алгоритму вірогідність (у відсотках) того, що елемент, який шукається, знаходиться у меншій частині приблизно дорівнює

$$\frac{1}{3} \cdot 100\%$$

Через це можна порахувати середню асимптотику роботи цього алгоритму.

$$n \cdot \left(\left(\frac{2}{3}\right)^x \cdot \left(\frac{1}{3}\right)^y\right) = 1$$

$$x + y = k$$

$$n = 1.5^x \cdot 3^y$$

Так як у 66% випадків програма буде запускатися із більшої частини, то

$$x = 2 \cdot y$$

$$k = 3 \cdot y$$

$$n = 1.5^{(2 \cdot y)} \cdot 3^y$$

$$n = \left(\frac{3}{2}\right)^{(2 \cdot y)} \cdot 3^y$$

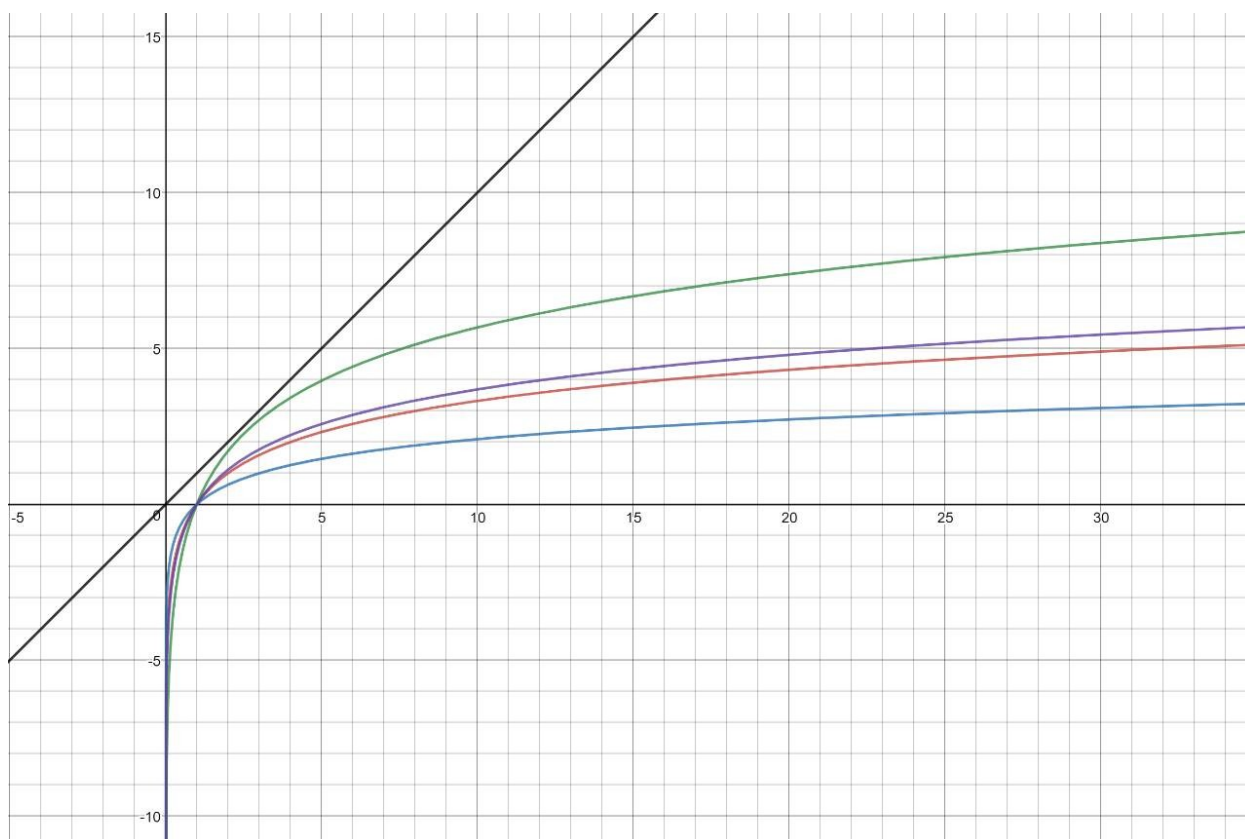
$$n = \left(\frac{27}{4}\right)^y$$

$$n = 6.75^y$$

$$y = \log_{6.75} n$$

$$k = 3 \cdot y = 3 \cdot \log_{6.75} n$$

Порівняння асимптотики алгоритмів



Чорний колір – асимптотика виконання алгоритму лінійного пошуку

Зелений колір – максимальна асимптотика виконання алгоритму А

Фіолетовий колір – середня асимптотика виконання алгоритму А

Червоний колір – асимптотика бінарного пошуку

Синій колір – найменша асимптотика виконання алгоритму А

На графіку (мал. 2.1) видно, що середній час роботи цього алгоритму, буде трохи більшим за час роботи бінарного пошуку. Тобто, при великому числі тестів та великій кількості чисел у кожному тесті, цей алгоритм у середньому буде працювати приблизно як і бінарний пошук (за часом), але все ж таки трохи довше. Тому цей алгоритм слід використовувати тільки тоді, коли число досить мале, та якщо тестів буде мало, але чисел у кожному з них багато.

Зафіксуємо час роботи цього алгоритму у кількох випадках. Колі було всього 100 тестів, найкращим результатом була рівно 1 секунда виконання

програми, що менше за мінімальний час роботи бінарного пошуку. А найгіршим результатом було 1.229 секунди, що більше за максимальний час виконання бінарного пошуку. Але частіше всього програма працювала від 1.05 секунди до 1.15. Це ще раз доводить, що використання алгоритму А на маленькій кількості тестів, буде доброю думкою.

Далі протестуємо цей алгоритм на 10000 тестів. На такому числі тестів цей алгоритм працював дещо по іншому. Після кількох запусків програма показала, що алгоритм працював у середньому 107.886 секунди, що менше за середній час виконання бінарного пошуку на 10000 тестах.

2.2 Власний алгоритм пошуку Б (Алгоритм невідомого пошуку)

У розділі 2 було описано про різну швидкість виконання програм. І було зрозуміло, що алгоритми, які працюють довше, ніж за логарифмічний час не підходять. У алгоритму А максимальний, мінімальний та середній час виконання є логарифмічний час. Але алгоритми, асимптотика яких може бути константою (наприклад 1 чи 2), ще не розглядалися. Розглянемо алгоритм, асимптотика якого може варіюватись від константи до лінійного часу.

Зробимо алгоритм Б, користуючись методом розбиття, тому що з усіх методів, цей є найефективнішим (але його не завжди можна застосувати (у нашому випадку можна)).

Алгоритм Б, як і два попередні, може працювати тільки з відсортованими масивами. Також його схожістю з попередніми є те, що алгоритм Б з легкістю може працювати з від'ємними числами (але нам це не знадобиться у програмі). Цей алгоритм написаний так само, як і два попередні, методом розбиття.

Алгоритм бінарного пошуку і алгоритм А поділяють масив на дві частини завжди в одному місці. Алгоритм бінарного пошуку це робить у середині масиву, а алгоритм А ділить масив на одній третині від його довжини. Алгоритм Б буде робити це у випадковому місці.

Алгоритм Б працює так. Він бере випадкове число від 1 до $n-1$ (де n це кількість чисел у масиві), а далі дивиться, якщо число яке алгоритм вибрав більше, ніж те, що шукається, то цей алгоритм запускається зліва від обраного елемента, якщо обраний елемент менший за той, що шукається, то алгоритм запускає себе праворуч від обраного елемента, а якщо обраний елемент дорівнює шуканому, то алгоритм запам'ятовує все, що треба (місце знаходження, кількість ітерацій і т. д.), і завершує свою роботу. Така реалізація алгоритму є рекурсивною, тому що алгоритм запускає сам себе, отже він рекурсивний.

При найгіршому розвитку подій максимальна асимптотика алгоритму Б буде виглядати наступним чином. Якщо на кожній ітерації алгоритм буде обирати позицію межі двох частин масиву одразу після першого елемента, а шуканий елемент буде у самому кінці, то цей алгоритм зробить n ітерацій. Отже його асимптотика буде дорівнювати $O(n)$. Тобто у цьому випадку він буде працювати, як і алгоритм лінійного пошуку (за часом).

Тепер розглянемо найменшу асимптотику алгоритму Б. Наприклад, коли шуканий елемент буде дорівнювати першому елементу у масиві, та межа поділу двох частин масиву буде якраз біля першого елемента масиву. Тоді знадобиться лише одна ітерація, отже асимптотика буде дорівнювати $O(1)$, а це константа.

Були розглянуті випадки коли алгоритм має найбільшу асимптотику, та коли алгоритм має найменшу асимптотику. Залишається лише одне питання. За яку середню асимптотику буде працювати цей алгоритм?

Розглянемо випадок, коли у масиві кількість елементів дорівнює нескінченності (або дуже великому числу) та кількість тестів також нескінченна. Зараз подивимось тільки на першу ітерацію кожного тесту. Так як всього нескінченна кількість тестів, то кількість разів, коли номер місця поділу двох частин дорівнюватиме одному, буде рівне кількості тестів, в яких номер поділу двох частин масиву дорівнювати двом, и так далі $n-1$ до n . Нехай ця кількість буде дорівнювати X , а кожне таке місце поділу будемо називати середнім. Подивимось коли середнє місце дорівнює одному і $n-1$, тоді середнє

арифметичне буде дорівнювати $\frac{n}{2}$, так само для $2i$, $n-2$ для $3i$ $n-3$, і так далі... тобто вийде $\frac{(n-2)}{2}$ чисел $\frac{n}{2}$, а середнє арифметичне всіх цих чисел буде дорівнювати:

$$\frac{\frac{n-1}{2} \cdot \frac{n}{2}}{\frac{n-2}{2}} = \frac{n}{2}$$

Тобто в середньому на першій ітерації буде братися елемент так само, як і у алгоритмі бінарного пошуку. А так як реалізація цього алгоритму є рекурсивною, то на кожній ітерації алгоритм, грубо кажучи, запускає першу ітерацію тільки з іншими параметрами. Отже на кожній ітерації середній елемент в середньому буде знаходитись на позиції $\frac{n}{2}$. Отже можна порахувати його середню асимптотику (k – відповідь, n – кількість елементів у масиві):

$$\frac{n \cdot 1}{2^k} = 1$$

$$n = 2^k$$

$$k = \log_2 n$$

Отже середня асимптотика буде такою ж самою як і у бінарного пошуку ($\log_2 n$).

Порівняймо алгоритм Б із алгоритмом бінарного пошуку. На 100 тестах алгоритм видає кожен раз різні значення. Найменше значення дорівнює 0.954 секунди, що набагато менше за найменший час виконання усіх вищеописаних алгоритмів. А найбільше значення було 1.301 секунди, що є найбільшим значенням серед усіх вищеописаних алгоритмів (окрім лінійного пошуку).

Тепер запустимо програму для 10000 тестів. Середній результат становить 107.94 секунди, що знову менше ніж працює бінарний пошук, та майже дорівнює часу виконання алгоритму А.

Як часто асимптотика алгоритму Б буде дорівнювати асимптотиці лінійного пошуку або 1? Якщо масив має n елементів, то вірогідність того, що

асимптотика буде $O(1)$, буде дорівнювати (у відсотках) $\frac{1}{n} \cdot 100\%$. І при $n = 100$, ця вірогідність дорівнює 1%, при $n = 1000$, ймовірність буде дорівнювати 0,1%, а при $n = 10000$, вірогідність буде 0,01%, тобто цього майже ніколи не буде. Тепер розглянемо яка ймовірність того, що алгоритм Б буде працювати за лінійний час. Тоді останній елемент має бути у кінці ($1/n$), і на кожній ітерації ми маємо брати протилежний елемент($1/i$). Тобто ймовірність буде дорівнювати:

$$\frac{1}{n} \cdot \frac{1}{n-1} \cdot \frac{1}{n-2} \cdot \frac{1}{n-3} \cdot \dots \cdot \frac{1}{1} = \frac{1}{n!} \cdot 100\%$$

При $n = 10$, ймовірність того, що алгоритм Б буде працювати за лінійний час буде: $100 \frac{\%}{10!} = \frac{100\%}{3628800} = 0.00003\%$, тобто таких випадків майже ніколи не буде (можуть бути при малих масивах, наприклад 5).

Цей алгоритм добре підходить для великих тестів та масивів великого розміру. Моя програма використовує масив малого розміру, отже асимптоматика алгоритму буде варіюватись між значеннями, різниця яких дуже велика.

РОЗДІЛ 3

ПОРІВНЯЛЬНИЙ АНАЛІЗ АЛГОРИТМІВ

3.1. Розробка тестів та тестування алгоритмів

Для перевірки дієздатності кожного з алгоритмів, щоб не писати багато тестів вручну, була розроблена програма, яка створює тести. Для початку треба розібратись що таке тестування.

Тестування – це процес перевірки правильності роботи усієї програми або її складових частин. [2]

Програма, яка створює тести для вищеописаних алгоритмів спочатку зчитує число (кількість тестів) `col`, після цього запускає цикл, який виконається `col` разів. На кожній ітерації циклу буде створений масив, почне рахуватись час виконання `g`-го тесту. Далі йде ще один цикл з `n` ітерацій, де `n` – випадкове число (розмір масиву). Далі на кожній ітерації другого циклу буде заповнюватись елемент масиву наступним чином. Створемо випадкове число, і якщо воно націло ділиться на 3, то програма присвоює наступному елементу масиву будь яке значення більше, ніж минуле, а якщо випадкове число не ділиться націло на 3, то програма наступному елементу присвоює значення минулого елементу. Тобто у середньому буде йти по 2 різні елементи підряд, але середню кількість елементів, які будуть йти підряд можна легко збільшити, або зменшити. Якщо дивитись на випадкове число на під модулем, більшим, наприклад, тоді середнє число однакових чисел, що йдуть підряд, збільшиться.

Також потрібен елемент, який буде шуканим. Перед другим циклом створюється змінна та їй присвоюється випадкове число. А на кожній ітерації другого циклу створюється випадкове число, і якщо воно ділиться на 100 без залишку, то елементу, який буде шуканим присвоюється `i`-й елемент масиву. Збільшуючи чи зменшуючи значення по модулю, якого ми беремо, ми збільшуємо ймовірність того, що ми будемо шукати число, яке є у масиві.

Також кожен тест записується у текстовий файл `output.txt`, тому можна буде перевірити правильність роботи алгоритму самому. І результат роботи алгоритму

на кожному тесті записується ще у один текстовий файл result.txt, щоб можна було запам'ятати як алгоритм веде себе в тій чи іншій ситуації.

Найголовніший тест, по якому була порівняна робота алгоритмів – це швидкість їх роботи, тому у самому початку замірюється час початку роботи алгоритму, а в кінці програми замірюється час кінця роботи програми. Різниця між цими двома значеннями і є часом виконання програми.

3.2. Порівняльна характеристика часу роботи алгоритмів

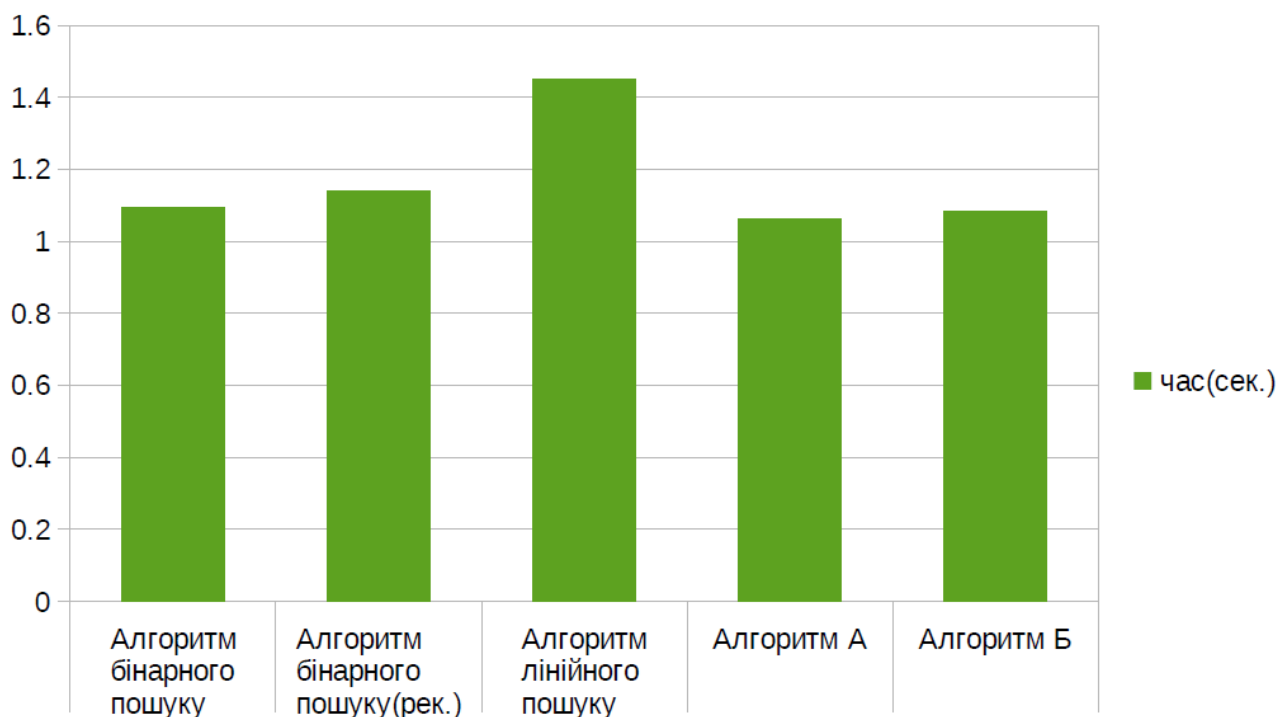
Зараз розглянемо, як результати тестування алгоритмів відрізняються один від одного у наступній таблиці (табл.3.1):

Таблиця.3.1

Порівняльна таблиця швидкості алгоритмів на 100 тестах

Назва алгоритму	Час роботи						
	на першому тесті	на другому тесті	на третьому тесті	на четвертому тесті	на п'ятому тесті	на шостому тесті	середній
Алгоритм бінарного пошуку	1,110	1,116	1,035	1,105	1,08	1,113	1,093
Алгоритм бінарного пошуку(рек.)	1,172	1,18	1,104	1,046	1,205	1,137	1,141
Алгоритм лінійного пошуку	1,367	1,46	1,392	1,483	1,583	1,418	1,451
Алгоритм А	1,087	1,048	1,021	1,047	1,09	1,089	1,064
Алгоритм Б	1,029	1,004	1,116	1,162	1,114	1,074	1,083

Порівняльна діаграма алгоритмів на 100 тестах



Коли кількість тестів не перевищувала значення сотні (табл. 3.1), всі алгоритми у середньому працювали не довше, ніж півтори секунди. На цьому прикладі видно, що при малих тестах взагалі байдуже який використовувати алгоритм. Ще велику схожість часу праці алгоритмів можна пояснити тим, що чим ближче кількість тестів до нуля, тим ближче X до нуля, тим менше Y (X дорівнює кількості тестів, а $Y = f(X)$, де $f(X)$ – це асимптотика алгоритму). Також можна відмітити, що алгоритм бінарного пошуку у рекурсивній реалізації працює довше всіх алгоритмів, які працюють за логарифмічний час.

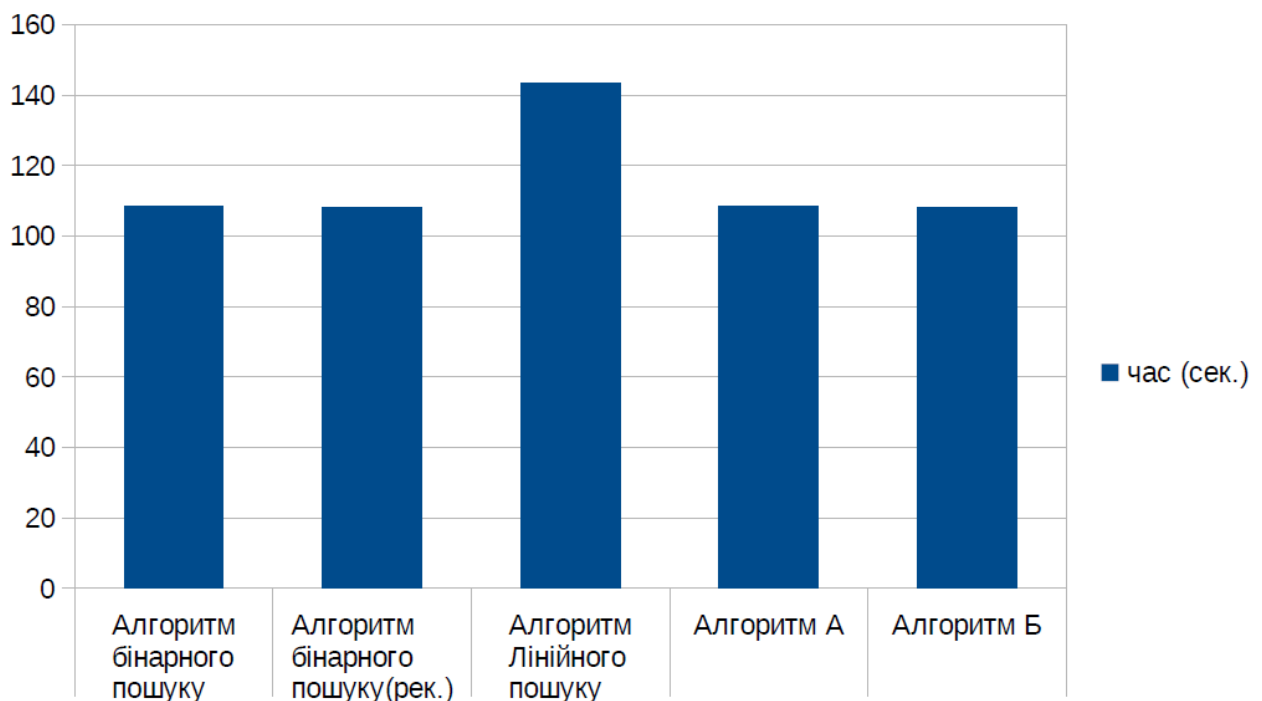
Таблиця.3.2

Порівняльна таблиця швидкості алгоритмів на 10000 тестах

Назва алгоритму	Час роботи						
	на першому тесті	на другому тесті	на третьому тесті	на четвертому тесті	на п'ятому тесті	на шостому тесті	середній
Алгоритм бінарного пошуку(10000)	108,986	108,081	108,207	108,11	109,02	107,995	108,400
Алгоритм бінарного пошуку(рек.)(10000)	108,079	107,525	108,562	108,326	108,16	108,24	108,149
Алгоритм лінійного пошуку(10000)	141,276	144,028	140,498	142,11	146,182	145,415	143,252
Алгоритм А(10000)	108,427	108,78	109,24	107,14	108,191	108,285	108,344
Алгоритм Б(10000)	107,185	107,512	109,259	108,14	108,261	108,6718	108,171

мал.3.2.

Порівняльна діаграма на 10000 тестах



У випадку коли кількість тестів 10000 (табл.3.2), ми маємо іншу картину. Можна побачити, що всі алгоритми працюють у середньому за один і той самий час (дивись малюнок 2.1), окрім алгоритму лінійного пошуку. Як і в випадку зі 100 тестами алгоритм лінійного пошуку працює довше всіх, тому цей алгоритм краще використовувати тільки коли тестів не більше, ніж 100. Хочу також відмітити цікавий момент. Коли тестів було 100, алгоритм бінарного пошуку з рекурсивною реалізацією працював довше, ніж всі інші алгоритми з логарифмічним часом виконання. А коли тестів стало 10000, алгоритм ділення пополам з рекурсивною реалізацією працював швидше, ніж всі інші алгоритми. Довше всіх (після алгоритму лінійного пошуку) на 10000 тестах працював алгоритм бінарного пошуку з ітераційною реалізацією. Також алгоритм Б в обох випадках (і зі 100 тестами, і з 10000 тестами) працював другим за швидкістю. Хоча він у середньому має працювати, як і бінарний пошук (за $O(\log_2 n)$).

ВИСНОВКИ

1. З вищесказаного можна зробити висновок, що алгоритми пошуку є дуже важливими. Вони бувають різні, в залежності від виду оброблюваних даних. Так, при їх програмуванні, інколи буває простіше і вигідніше з урахуванням процесорного часу, часу проведення пошуку, кількості використаних операцій, провести сортування масиву даних, а вже потім проводити пошук за допомогою одного із методів пошуку. Проте, інколи, сортування масиву є зайвим, оскільки в деяких випадках витрати часу на сортування масиву не перебивають затрат виконаних на пошук масиву простим, лінійним перебором елементів масиву. Аналогічно, не завжди можна використати простий перебір елементів, як от наприклад, коли необхідно визначити елемент, який найбільш наближений до середнього арифметичного у масиві.

2. При малому масиві елементів в малому кількості тестів можна використовувати будь який з розглянутих алгоритмів.

3. При великому масиві чи великої кількості тестів слід використовувати алгоритми з логарифмічною асимптотикою (тобто не можна використовувати лінійний пошук).

4. Алгоритм А дає можливість програмі виконати меншу кількість дій, ніж у бінарному пошуку. Що дозволить зменшити середній час виконання програм у більшості випадків.

5. Алгоритм Б має ймовірність працювати за константний час, чого не може зробити жоден з існуючих алгоритмів пошуку.

6. Зробивши порівняльну характеристику часу роботи алгоритмів, можемо зробити висновок, що алгоритми, які працюють за логарифмічний час, при великій кількості тестів працюють приблизно однаково (за один і той самий час), а при маленьких тестах, їх час роботи дещо відрізняється.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бхаргава А. Грокаємо алгоритми. Ілюстрований посібник для програмістів і тих, хто цікавиться. / А. Бхаргава – СПб.: Пітер, 2018. – 288 с.
2. Кнут Д.Є. Мистецтво програмування , том 3. Пошук і сортування, 3-е вид.: Пер. з англ.: навч. посіб. / Д.Є. Кнут; – М.: Видавничий дім "Вільямс", 2000. – 750 с.

ІНТЕРНЕТ-РЕСУРСИ

<http://www.cppstudio.com/post/446/>

<http://www.interface.ru/home.asp?artId=35216>

<http://www.matematikam.ru/calculate-online/grafik.php>