МІНІСТЕРСТВО НАУКИ І ОСВІТИ УКРАЇНИ

Херсонський Державний Педагогічний Університет

Фізико-математичний факультет

Кафедра інформаційних технологій

Швидкі алгоритми сортування

**Курсова робота**

Виконавець

Керівник

**Херсон 2001**

Зміст

Вступ 3

1. Аналіз швидких алгоритмів сортування 6

1.1. Сортування деревом 6

1.2. Пірамідальне сортування 9

1.3 Швидке сортування Хоара 12

1.4 Метод цифрового сортування 14

2. Практична реалізація мовою Паскаль швидких алгоритмів сортування 16

Висновки 22

Список використаних джерел 24

# **Вступ**

В наш час нові інформаційні технології посідають дуже важливе місце не лише в спеціалізованих, але й в повсякденних сферах життя. Комп’ютери застосовуються в бізнесі, менеджменті, торгівлі, навчанні та багатьох інших сферах діяльності людини.

Комп’ютерні технології дуже зручні для виконання різноманітних операцій, але в різних сферах застосування ці операції різні. Тому, кожна окрема галузь, яка використовує специфічні технічні засоби, потребує своїх власних програм, які забезпечують роботу комп’ютерів.

Розробкою програмного забезпечення займається така галузь науки, як програмування. Вона набуває все більшого й більшого значення останнім часом, адже з кожним днем комп’ютер стає все більш необхідним, все більш повсякденним явищем нашого життя. Адже обчислювальна техніка минулих років вже майже повністю вичерпала себе і не задовольняє тим потребам, що постають перед людством.

Таким чином, нові інформаційні технології дуже актуальні в наш час і потребують багато уваги для подальшої розробки та вдосконалення. Поряд з цим, велике значення має також і програмування, яке є одним із фундаментальних розділів інформатики і тому не може залишатись осторонь.

Програмування містить цілу низку важливих внутрішніх задач. Однією з найбільш важливих таких задач для програмування є задача сортування. Під сортуванням звичайно розуміють перестановки елементів будь-якої послідовності у визначеному порядку. Ця задача є однією з найважливіших тому, що її метою є полегшення подальшої обробки певних даних і, насамперед, задачі пошуку. Так, одним з ефективних алгоритмів пошуку є бінарний пошук. Він працює швидше ніж, наприклад, лінійний пошук, але його можливо застосовувати лише за умови, що послідовність вже упорядкована, тобто відсортована.

Взагалі, відомо, що в будь-якій сфері діяльності, що використовує комп’ютер для запису, обробки та збереження інформації, усі дані зберігаються в базах даних, які також потребують сортування. Певна впорядкованість для них дуже важлива, адже користувачеві набагато легше працювати з даними, що мають певний порядок. Так, можна розташувати всі товари по назві або відомості про співробітників чи студентів за прізвищем або роком народження, тощо.

Задача сортування в програмуванні не вирішена повністю. Адже, хоча й існує велика кількість алгоритмів сортування, все ж таки метою програмування є не лише розробка алгоритмів сортування елементів, але й розробка саме ефективних алгоритмів сортування. Ми знаємо, що одну й ту саму задачу можна вирішити за допомогою різних алгоритмів і кожен раз зміна алгоритму приводить до нових, більш або менш ефективних розв’язків задачі. Основними вимогами до ефективності алгоритмів сортування є перш за все ефективність за часом та економне використання пам’яті. Згідно цих вимог, прості алгоритми сортування (такі, як сортування вибором і сортування включенням) не є дуже ефективними.

Алгоритм сортування обмінами, хоча і завершує свою роботу (оскільки він використовує лише цикли з параметром і в тілі циклів параметри примусово не змінюються) і не використовує допоміжної пам’яті, але займає багато часу. Навіть, якщо внутрішній цикл не містить жодної перестановки, то дії будуть повторюватись до тих пір, поки не завершиться зовнішній цикл.

Алгоритм сортування вибором ефективніше сортування обмінами за критерієм М(n), тобто за кількістю пересилань, але також є не дуже ефективним. З цих причин було розроблено деякі нові алгоритми сортування, що отримали назву швидких алгоритмів сортування. Це такі алгоритми, як сортування деревом, пірамідальне сортування, швидке сортування Хоара та метод цифрового сортування.

Метою нашої дослідницької роботи є ознайомлення з цими швидкими алгоритмами сортування, спроба проаналізувати їх і висвітлити кожен з них і написати програму, яка б виконувала сортування деякої послідовності за допомогою різних швидких алгоритмів сортування.

# **1. Аналіз швидких алгоритмів сортування**

## 1.1. Сортування деревом

Алгоритм сортування деревом ТreeSort власне кажучи є поліпшенням алгоритму сортування вибором. Процедура вибору найменшого елемента удосконалена як процедура побудови т.зв. сортуючого дерева. Сортуюче дерево - це структура даних, у якій представлений процес пошуку найменшого елемента методом попарного порівняння елементів, що стоять поруч. Алгоритм сортує масив у два етапи.

1. I етап : побудова сортуючого дерева;
2. II етап : просівання елементів по сортуючому дереву.

Розглянемо приклад: Нехай масив A складається з 8 елементів (мал. 1, 1-а рядок). Другий рядок складається з мінімумів елементів першого рядка, які стоять поруч. Кожний наступний рядок складений з мінімумів елементів, що стоять поруч, попереднього рядка.

**a1**

**a2**

**a3**

**a4**

**a5**

**a6**

**a7**

**a8**

**a2**

**a3**

**a5**

**a8**

**a3**

**a5**

**a5**

a2 = min(a1,a2)

a3 = min(a3,a4)

a5 = min(a5,a6)

a8 = min(a7,a8)

a3 = min(a2,a3)

a5 = min(a5,a8)

a5 = min(a3,a5)

**мал 1**

Ця структура даних називається сортуючим деревом. У корені сортуючого дерева розташований найменший елемент. Крім того, у дереві побудовані шляхи елементів масиву від листів до відповідного величині елемента вузла - розгалуження. (На мал.1 шлях мінімального елемента **a5** - від листа **a5** до кореня відзначений товстою лінією.)

Коли дерево побудоване, починається етап просівання елементів масиву по дереву. Мінімальний елемент пересилається у вихідний масив B і усі входження цього елемента в дереві заміняються на спеціальний символ **M.**

**a1**

**a2**

**a3**

**a4**

**M**

**a6**

**a7**

**a8**

**a2**

**a3**

**M**

**a8**

**a3**

**M**

**M**

**a5**

В

**мал 2**

Потім здійснюється просівання елемента уздовж шляху, відзначеного символом M, починаючи з листка, сусіднього з M (На мал 2. униз) і до кореня. Крок просівання - це вибір найменшого з двох елементів, що зустрілися на шляху до кореня дерева і його пересилання у вузол, відзначений M. Просівання 2-го елемента показано на мал 3. (Символ М більше, ніж будь-який елемент масиву).

**a3**

**a4**

**M**

**a6**

**a7**

**a8**

**a3**

**a6**

**a8**

**a3**

**a6**

**a3**

**a1**

**a2**

**a2**

**a5 a3**

a6 = min(M, a6)

a6 = min(a6, a8)

a3 = min(a3, a6)

b2 := a3

Просівання елементів відбувається доти, поки весь вихідний масив не буде заповнений символами M, тобто n раз:

For I := 1 to n do begin

Відзначити шлях від кореня до листка символом M;

Просіяти елемент уздовж відзначеного шляху;

B[I] := корінь дерева

end;

Обґрунтування правильності алгоритму очевидно, оскільки кожне чергове просівання викидає в масив **У** найменший з елементів масиву А, що залишилися.

Сортуюче дерево можна реалізувати, використовуючи або двовимірний масив, або одномірний масиві ST[1..N], де N = 2n-1 (див. наступний розділ). Оцінимо складність алгоритму в термінах M(n), C(n). Насамперед відзначимо, що алгоритм TreeSort працює однаково на усіх входах, так що його складність у гіршому випадку збігається зі складністю в середньому.

Припустимо, що n - ступінь 2 (n = 2l). Тоді сортуюче дерево має l + 1 рівень (глибину l). Побудова рівня I вимагає n / 2I порівнянь і пересилань. Таким чином, I-ий етап має складність:

**C1(n) = n/2+n/4+ ... + 2+1 = n-1, M1(n) = C1(n) = n - 1**

Для того, щоб оцінити складність II-го етапу З2(n) і M2(n) помітимо, що кожен шлях просівання має довжину l, тому кількість порівнянь і пересилань при просіванні одного елемента пропорційно l. Таким чином, M2(n) = O(l n), C2(n) = O(l n).

Оскільки l = log2n, M2(n)=O(n log2 n)), C2(n)=O(n log2 n), Але З(n) = C1(n) + C2(n), M(n) = M1(n) + M2(n). Тому що C1(n) < C2(n), M1(n) < M2(n), остаточно одержуємо оцінки складності алгоритму TreeSort за часом:

**M(n) = O(n log2 n), C(n) = O(n log2 n),**

У загальному випадку, коли n не є ступенем 2, сортуюче дерево будується трохи інакше. “Зайвий” елемент (елемент, для якого немає пари) переноситься на наступний рівень. Легко бачити, що при цьому глибина сортуючого дерева дорівнює [log2 n] + 1. Удосконалення алгоритму II етапу очевидно. Оцінки при цьому змінюють лише мультиплікативні множники. Алгоритм TreeSort має істотний недолік: для нього потрібно додаткова пам'ять розміру 2n - 1.

## 1.2. Пірамідальне сортування

Алгоритм пірамідального сортування HeapSort також використовує представлення масиву у виді дерева. Цей алгоритм не вимагає допоміжних масивів, сортуючи “на місці”. Розглянемо спочатку метод представлення масиву у виді дерева:

Нехай A[1 .. n] - деякий масив. Зіставимо йому дерево, використовуючи наступні правила:

1.A[1] - корінь дерева ;

**A[2i]**

**A[2i+1]**

**A[i]**

2.Якщо A[i] - вузол дерева і 2i ≤ n,

то A[2\*i] - вузол - “лівий син” вузла A[i]

3.Якщо A[i] - вузол дерева і 2i + 1 ≤ n,

то A[2\*i+1] - вузол - “правий син” вузла A[i]

Правила 1-3 визначають у масиві структуру дерева, причому глибина дерева не перевершує [log2 n] + 1. Вони ж задають спосіб руху по дереву від кореня до листків. Рух вгору задається правилом 4:

4.Якщо A[i] - вузол дерева і i > 1,

то A[i mod 2] - вузол - “батько” вузла A[i];

Приклад: Нехай A = [45 13 24 31 11 28 49 40 19 27] - масив. Відповідне йому дерево має вид:

**45**

**13**

**24**

**31**

**11**

**28**

**49**

**40**

**19**

**27**

Зверніть увагу на те, що всі рівні дерева, за винятком останнього, цілком заповнені, останній рівень заповнений ліворуч і індексація елементів масиву здійснюється вниз і праворуч. Тому дерево упорядкованого масиву відповідає наступним властивостям:

A[i] ( A[2\*i], A[i] ( A[2\*i+1], A[2\*i] ( A[2\*i+1].

Як це не дивно, алгоритм HeapSort спочатку будує дерево, що відповідає прямо протилежним співвідношенням:

A[i] ≥ A[2\*i], A[i] ≥ A[2\*i+1]

а потім змінює місцями A[1] (найбільший елемент) і A[n].

Як і TreeSort, алгоритм HeapSort працює в два етапи:

I. Побудова сортуючого дерева;

II. Просівання елементів по сортуючому дереву.

Дерево, що представляє масив, називається сортуючим, якщо виконуються умови (6). Якщо для деякого i ця умова не виконується, будемо говорити, що має місце (сімейний) конфлікт у трикутнику i.

Як на I-ом, так і на II-ому етапах елементарна дія алгоритму полягає в вирішенні сімейного конфлікту: якщо найбільший із синів більше, ніж батько, то переставляються батько і цей син (процедура ConSwap).

У результаті перестановки може виникнути новий конфлікт у тому трикутнику, куди переставлений батько. У такий спосіб можна говорити про конфлікт (роду) у піддереві з коренем у вершині i. Конфлікт роду вирішується послідовним вирішенням сімейних конфліктів проходом по дереву вниз. (На мал шлях вирішення конфлікту роду у вершині 2 відзначений). Конфлікт роду вирішено, якщо прохід закінчився (i > n div 2), або ж в результаті перестановки не виник новий сімейний конфлікт (процедура Conflict).

Procedure ConSwap(i, j : Integer);

Var b : Real;

Begin

If a[i] < a[j] then begin

b := a[i]; a[i] := a[j]; a[j] := b

end

End;

Procedure Conflict(i, k : Integer);

Var j : Integer;

Begin

j := 2\*i;

If j = k

then ConSwap(i, j)

else if j < k then begin

if a[j+1] > a[j] then j := j + 1;

ConSwap(i, j); Conflict(j, k)

end

End;

I етап – побудова сортуючого дерева - оформимо у виді рекурсивної процедури, використовуючи визначення:

Якщо ліве і праве піддерева T(2i) і T(2i+1) дерева T(i) є сортуючими, то для перебудови T(i) необхідно вирішити конфлікт роду в цьому дереві.

Procedure SortTree(i : Integer);

begin

If i <= n div 2 then begin

SortTree(2\*i); SortTree(2\*i+1); Conflict(i, n)

end

end;

На II-ом етапі - етапі просівання - для k від n до 2 повторюються наступні дії:

1.Переставити A[1] і A[k];

2.Побудувати сортуюче дерево початкового відрізка масиву A[1..k-1], усунувши конфлікт роду в корені A[1]. Відзначимо, що 2-а дія вимагає введення в процедуру Conflict параметра k.

Program HeapSort;

Const n = 100;

Var A : Array[1..n] of real;

k : Integer;

{процедури ConSwap, Conflict, SortTree, введення, виведення}

Begin

{ введення }

SortTree(1);

For k := n downto 2 do begin

ConSwap(k, 1); Conflict(1, k - 1)

end;

{ виведення }

End.

Нескладно підрахувати, що С**(n) = O( n log2 n ),**  М**(n) = O( n log2 n )**

## 

## 1.3 Швидке сортування Хоара

Удосконаливши метод сортування, який грунтується на обмінах, К. Хоар запропонував алгоритм QuickSort сортування масивів, що дає на практиці відмінні результати і дуже просто програмується. Автор назвав свій алгоритм швидким сортуванням.

Ідея К. Хоара полягає в наступному:

1 Виберемо деякий елемент x масиву A випадковим образом;

2.Переглядаємо масив у прямому напрямку (i = 1, 2,...), шукаючи

в ньому елемент A[i] не менший за x;

3.Переглядаємо масив у зворотньому напрямку (j = n, n-1,..),

шукаючи в ньому елемент A[j] не більший за x;

4.Змінюємо місцями A[i] і A[j];

Пункти 2-4 повторюємо доти, поки i < j;

У результаті такого зустрічного проходу початок масиву A[1..i] і кінець масиву A[j..n] виявляються розділеними “бар'єром” x: A[k] ( x при k < i, A[k] ( x при k > j , причому на поділ ми затратимо не більш n/2 перестановок. Тепер залишилося проробити ті ж дії з початком і кінцем масиву, тобто застосувати їх рекурсивно.

Таким чином, описана нами процедура Hoare залежить від параметрів k та m - початкового і кінцевого індексів відрізка масиву, який обробляється.

Procedure Swap(i, j : Integer);

Var b : Real;

Begin

b := a[i]; a[i] := a[j]; a[j] := b

End;

Procedure Hoare(L, R : Integer);

Var left, right : Integer;

x : Integer;

Begin

If L < R then begin

x := A[(L + R) div 2]; {вибір бар'єра x}

left := L; right := R ;

Repeat {зустрічний прохід}

While A[left] < x do Inc(left); {перегляд уперед}

While A[right] > x do Dec(right); {перегляд назад}

If left <= right then begin

Swap(left, right); {перестановка}

Inc(left); Dec(right);

end

until left > right;

Hoare(L, right); {сортуємо початок}

Hoare(left, R) {сортуємо кінець}

end

End;

Program QuickSort;

Const n = 100;

Var A : array[1..n] of Integer;

{ процедури Swap, Hoare, введення і висновку }

Begin

Inp; Hoare(1, n); Out

End.

Аналіз складності алгоритму в середньому, що використовує гіпотезу про рівну імовірність усіх входів, показує, що:

**C(n) = O(n log2 n), M(n) = O(n log2 n)**

У гіршому випадку, коли в якості бар'єрного вибирається, наприклад, максимальний елемент підмассиву, складність алгоритму квадратична.

## 

## 1.4 Метод цифрового сортування

Іноді при розв’язанні задач типу задачі сортування можна використовувати особливості типу перетворюваних даних для одержання ефективного алгоритму. Розглянемо одну з таких задач - задачу про звертання підстановки.

Підстановкою безлічі 1..n назвемо двовимірний масив A[1..2, 1..n] виду

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1** | **2** | **..........** | **n-1** | **n** |
| **J1** | **j2** | **...........** | **jn-1** | **jn** |

у якому 2-ий рядок містить всі елементи відрізка 1..n. Підстановка B називається зворотньою до підстановки A, якщо B виходить з A сортуванням стовпчиків A у порядку зростання елементів 2-го рядка з наступною перестановкою рядків. Потрібно побудувати алгоритм обчислення зворотньої підстановки. З визначення випливає, що стовпчик [i, ji] підстановки A потрібно перетворити в стовпчик [ji , i] і поставити на ji-те місце в підстановці B.

{Type NumSet = 1..n;

Substitution = array[ 1..2, NumSet] of NumSet; }

Procedure Reverse ( Var A, B : Substitution);

Begin

For i := 1 to n do begin

B[A[2, i], 2] := i; B[A[2, i], 1] := A[2, i]

end

End;

Складність процедури Reverse лінійна, оскільки тіло арифметичного циклу складається з двох операторів присвоювання, в той час як стовпчики підстановки відсортовані.

# **2. Практична реалізація мовою Паскаль швидких алгоритмів сортування**

# 

Практичною метою нашої дослідницької роботи було написання мовою Pascal програми, яка б виконувала сортування будь-якої послідовності елементів. Для того, щоб продемонструвати роботу різних швидких алгоритмів сортування, ми залишили вибір конкретного алгоритму на розсуд користувача програми. Тобто, ми створили основну програму – меню, яка пропонує користувачеві три можливі варіанти швидких алгоритмів сортування: швидке сортування, сортування Хоара та сортування “злиттям”. Вибір певного алгоритму здійснюється за допомогою оператору “case of”, тобто натисканням клавіш клавіатури 1, 2 або 3. Також ми передбачили варіант, коли користувач програми натискає будь-яку іншу клавішу: в цьому випадку на екрані з’явиться повідомлення про помилку. Також, після проведення сортування за вибраним алгоритмом, користувач зможе продовжити роботу й випробувати інший алгоритм. Для цього потрібно натиснути клавіші клавіатури “у” або “д” або набрати слово “yes” чи “да” коли після завершення роботи одного з обраних алгоритмів сортування на екрані з’явиться запитання “Хотите ли вы продолжить работу с данной программой?”. Якщо ж користувач уже випробував усі алгоритми або за браком часу (або бажання) хоче завершити роботу з програмою, то йому достатньо буде лише натиснути на клавіатурі клавіші “n” або “н” чи набрати слова “no” або “нет” після того, як на екрані з’явиться зазначене вище запитання.

Програма виконує сортування послідовності за трьома алгоритмами сортування. Кожний окремий алгоритм представлений у вигляді окремої процедури.

Так, процедура Qsort виконує швидке сортування масиву, що вводиться. Під час роботи цього алгоритму відбувається аналіз даної послідовності одночасно у двох напрямках ( зліва-направо і зправа-наліво) . комп’ютер порівнює два елементи, що стоять поряд зліва. Якщо ці елементи стоять на своїх місцях, тобто перший з них є меншим за другий, то комп’ютер порівнює перший елемент з останнім. Якщо при порівнянні останній елемент виявиться меншим за перший, то комп’ютер виконає перестановку цих двох елементів. Такі дії будуть відбуватися до тих пір, поки індикатор, якій відповідає за ліву частину послідовності (в нашій процедурі це “i”) не перейде на праву частину, а індикатор, що відповідає за праву частину масиву (в нашій процедурі це “j”) не перейде на праву частину. Далі та ж сама процедура викликається рекурсивно. Тобто, якщо ліва частина вже відсортована, то ми викликаємо ту саму процедуру і комп’ютер виконує ті самі дії, але в параметрах процедури ми змінюємо ліву границю. Те саме відбувається, коли відсортована права частина масиву.

По суті цей алгоритм працює на основі алгоритму сортування обмінами, але цей алгоритм вважається швидким, оскільки перегляд послідовності відбувається у двох напрямках одночасно. Реалізовано ж цей алгоритм за допомогою оператору “if”, який відповідає за порівняння елементів, та пересилань.

Procedure \_Qsort (var a:mas; low, hi: byte);

Var i,j:byte;

begin

if hi> low then

begin i:= low;

j:= hi;

x:= a[i];

While i< j do

if a[i+1]<=x then

begin

a[i]:= a[i+1];

a[i+1]:=x;

i:= i+1;

end

else

begin

if a[j]<=x then

begin

y:=a[j];

a[j]:=a[i+1];

a[i+1]:=y;

end;

j:=j-1

end;

-Qsort (a, low, i-1);

-Qsort (a, i+1, hi);

end;

end;

Процедура Mergesort виконує сортування двох масивів “злиттям”. Тобто, спочатку перший масив сортується за допомогою процедури Qsort і другий масив сортується за допомогою цієї ж процедури. Потім два, вже відсортованих масиви з’єднуються в один. А саме, за допомогою оператору “if” ми порівнюємо перший елемент одного і перший елемент другого масивів. Якщо один з них менший, то ми відсилаємо його в третій масив, а індикатор пересуваємо на наступний елемент і знов порівнюємо їх. Знову менший з двох елементів пересилаємо в третій масив, а індикатор пересуваємо. Також ми передбачили варіант, коли елементи, що порівнюються – однакові. Тоді в третій масив по-черзі записуються обидва елементи, а індикатори зміщуються на наступні елементи в обох масивах. Якщо один з масивів виявився меншим за кількістю елементів ніж інший, то решта елементів довшого масиву просто пересилається до третього масиву в тій самій послідовності (адже вихідні масиви вже відсортовані раніше процедурою Qsort). Таким чином, в результаті ми отримуємо третій масив з впорядкованими елементами.

Procedure Mergesort;

Var i, j, k: byte;

Begin

i:=1;

j:=1;

for k:=1 to n\_c do c[k]:=0;

k:=1;

While ( i<= n\_a ) or ( j<=n\_b ) do

if i= n\_a+1 then

begin

c[k]:=b[j];

k:=k+1;

j:= j+1;

end

else

if i= n\_b+1 then

begin

c[k]:=a[i];

k:= k+1;

i:=i+1 ;

end

else

if a[i]= b[j] then

begin

c[k]:=b[j];

c[k+1]:= a[i];

k:=k+2;

i:=i+1;

j:=j+1

end

else

if a[i] > b[j] then

begin

c[k]:= b[j];

k:= k+1;

j:= j+1;

end

else

begin

c[k]:= a[i];

k:= k+1;

i:= i+1;

end

End;

Третя процедура описує алгоритм швидкого сортування Хоара. Це є удосконалений метод сортування, що базується на сортуванні обмінами. Тобто, спочатку ми обираємо деякий “бар’єр” (один з елементів масиву). Потім ми переглядаємо елементи, що стоять зліва від “бар’єра” і порівнюємо їх з ним. Коли ми знаходимо елемент, який є більшим за “бар’єр”, то ми починаємо перглядати масив з кінця, порівнюючи елементи з “бар’єром”. Якщо ми знайдемо в правій частині масиву елемент менший за “бар’єр”, то ми переставимо місцями елемент зліва (той, що більше за “бар’єр”)і елемент з права ( той, що менше) і продовжимо перегляд. Потім ми рекурсивно застосовуємо цю процедуру для того, щоб відсортувати початок і кінець масиву. Реалізуємо цей алгоритм за допомогою циклів “While” та “Repeat Until” та оператору “if”, який відповідає за порівняння.

Procedure HoareSearch ( var a:mas; L, R: Integer);

Var left, right, b, x: Integer;

Begin

if L < R then

begin

x:= a[( L+R) div 2];

left:= L;

right:= R;

Repeat

While a[ left] < x do left:= Succ(left);

While a[right] >x do right:= Pred(right);

If left >= right then

Begin

b:= a[left];

a[left]:= a[right];

a[right]:=b;

left:= Succ( left);

right:= Pred(right);

End;

Until left > right;

HoareSearch ( L, right);

HoareSearch (left, R);

end;

End;

Також у програмі представлена процедура, яка відповідає за введення масиву. Вона не винесена окремо в головну програму і користувач не побачить її на своєму екрані-меню. Він побачить лише ті три сортування, що написані у вигляді процедур.

В своїй роботі ми написали програму, що сортує масив за допомогою лише трьох алгоритмів сортування. Але можна створити процедури, які б містили алгоритми сортування деревом та пірамідального сортування. Це не вплине дуже суттєво на нашу програму. Потрібно буде лише додати ці процедури оператор “Case of” головної програми і користувач зможе обирати їх і використовувати їх так само, як і ті алгоритми, що вже були розглянуті нами в нашій дослідницькій роботі.

# **Висновки**

Отже, ми розглянули як працюють швидкі алгоритми сортування і спробували визначити їх складність.

Застосування того чи іншого алгоритму сортування для вирішення конкретної задачі є досить складною проблемою, вирішення якої потребує не лише досконалого володіння саме цим алгоритмом, але й всебічного розглядання того чи іншого алгоритму, тобто визначення усіх його переваг і недоліків.

Звичайно, необхідність застосування саме швидких алгоритмів сортування очевидна. Адже прості алгоритми сортування не дають бажаної ефективності в роботі програми. Але завжди треба пам’ятати й про те, що кожний швидкий алгоритм сортування поряд із своїми перевагами може містити і деякі недоліки.

Так, алгоритм сортування деревом, хоча й працює однаково на всіх входах (так, що його складність в гіршому випадку співпадає зі складністю в середньому), але цей алгоритм має і досить суттєвий недолік: для нього потрібна додаткова пам’ять розміром 2n-1.

Розглядаючи такий швидкий алгоритм сортування, як пірамідальне сортування, можна зазначити, що цей алгоритм ефективніший ніж попередній, адже він сортує “на місці” , тобто він не потребує додаткових масивів. Крім того, цей алгоритм (“ з точністю до мультиплікативної константи” (4, 74)) оптимальний: його складність співпадає з нижньою оцінкою задачі, тобто за критеріями C(n) та M(n) він має складність O(n log2 n), але містить складний елемент в умові. Тобто, в умові A[left] має бути строго менше ніж x , а A[right] - строго більше за x. Якщо ж замість “строго більше” та “строго менше” поставити знаки, що позначають “більше, або дорівнює” та “менше, або дорівнює”, то індекси left і right пробіжать увесь масив і побіжать далі. Вийти з цієї ситуації можна було б шляхом ускладнення умов продовження перегляду, але це б погіршило ефективність програми.

В нашій роботі ми розглянули деякі швидкі алгоритми сортування та їх реалізацію мовою Pascal, дослідили не лише переваги таких алгоритмів, ефективність їх використання, але й визначили деякі недоліки окремих алгоритмів, що заважають вживати їх для вирішення першої ліпшої задачі сортування.

Отже, головною задачею, яку має вирішити людина, яка повинна розв’язати задачу сортування – це визначення як позитивних, так і усіх негативних характеристик різних алгоритмів сортування, передбачення кінцевого результату. До того ж , треба враховувати головне – чи , можливо, цю задачу задовольнить один з класичних простих алгоритмів сортування.

# **Список використаних джерел**

1. Абрамов С. А., Зима Е. В. Начала программирования на языке Pascal. - М.: Наука, 1987.
2. Абрамов В. Г. Введение в язык Pascal: Учебное пособие для студентов вузов по специальности Прикладная математика. – М.: Наука, 1988.
3. Джонс Ж., Харроу К. Решение задач в системе Турбо-Паскаль/ Перевод с английского Улановой, Широкого. – М.: Финансы и статистика, 1991.
4. Зуев Е. А. Язык программирования Турбо Паскаль 6.0, 7.0. – М.: Радио и связь, 1993.
5. Культин Н. Б. Программирование в TurboPascal 7.0 и Delphi. - Санкт-петербург,1999.
6. Львов М. С., Співаковський О. В. Основи алгоритмізації та програмування. – Херсон, 1997.
7. Перминов О. Н. Программирование на языке Паскаль. – М.: Радио и связь, 1988.
8. Перминов О. Н. Язык программирования Pascal. – М.: Радио и свіязь,1989.
9. Турбо Паскаль 7.0 Издание 10-е стереотипное. – Санкт-Петербург: “Печатный Двор”, 1999.
10. Фаронов В. В. TurboPascal 7.0 . Начальный курс. – М.: “Нолидж”, 2000.