**Лабораторна робота № 3**

***Тема:* “Алгоритми сортування”.**

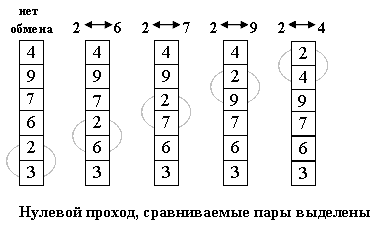
***Мета:*** Розлянути алгоритми сортування, дослідити доцільність застосування різних алгоритмів в конкретних випадках виходячи з їх ефективності.

**Сортування бульбашкою**

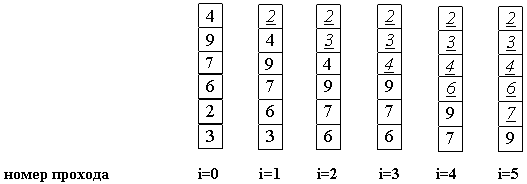
Розташуємо масив зверху вниз, від нульового елементу - до останнього.

Ідея методу: крок сортування полягає в проході від низу до верху по масиву. По дорозі переглядаються пари сусідніх елементів. Якщо елементи деякої пари знаходяться в неправильному порядку, то міняємо їх місцями.

Після нульового проходу по масиву "вгорі" виявляється "найлегший" елемент - звідси аналогія з бульбашкою. Наступний прохід робиться до другого зверху елементу, таким чином другий за величиною елемент піднімається на правильну позицію...



Робимо проходи по нижній частини масиву, що зменшується, до тих пір, поки в ній не залишиться тільки один елемент. На цьому сортування закінчується, оскільки послідовність впорядкована за зростанням.



*{ Сортуються записи типу item по ключу item.key }*

*{ для допоміжних змінних використовується тип index }*

*procedure BubbleSort;*

*var i,j: index; x:item;*

*begin for i:=2 to n do*

*begin for j:=n downto i do*

*if а[j-1].key > а[j].key then*

*begin x:=a[j-1]; а[j-1]:=a[j]; а[j]:=x;*

*end;*

*end;*

*end;*

Середнє число порівнянь і обмінів має квадратичний порядок зростання: О(n2), звідси можна зробити висновок, що алгоритм бульбашки дуже повільний і малоефективний. Проте, у нього є величезний плюс: він простий і його можна по-всякому покращувати.

Якісно інше поліпшення алгоритму можна одержати з наступного спостереження. Хоча легка бульбашка знизу підніметься вгору за один прохід, важкі бульбашки опускаються з мінімальною швидкістю: один крок за ітерацію. Отже масив 2 3 4 5 6 1 буде відсортований за 1 прохід, а сортування послідовності 6 1 2 3 4 5 вимагає 5 проходів.

Щоб уникнути подібного ефекту, можна міняти напрям слідуючих один за іншим проходів. Отриманий алгоритм іноді називають *шейкерним* сортуванням.

*procedure ShakerSort;*

*var j,k,l,r: index; x: item;*

*begin l:=2; r:=n; k:=n;*

*repeat*

*for j:=r downto l do*

*if а[j-1].key > а[j].key then*

*begin x:=a[j-1]; а[j-1]:=a[j]; а[j]:=x;*

*k:=j;*

*end;*

*l:=k+1;*

*for j:=l to r do*

*if а[j-1].key > а[j].key then*

*begin x:=a[j-1]; а[j-1]:=a[j]; а[j]:=x;*

*k:=j;*

*end;*

*r:=k-1;*

*until l>r;*

*end;*

Аналіз алгоритму простої бульбашки:

Число порівнянь C = 1/2 (n^2 - n)

Числа пересилок: Mmin = 0, Mсереднє = 3/4 (n^2 - n), Mmax = 3/2 (n^2 - n).

Аналіз сортування шейкера:

Мінімальне число порівнянь Cmin = n-1

Середнє число проходів пропорційно n - k1\*,

Cреднее число порівнянь пропорційно 1/2 ( n^2 - n(k2+ln(n))).

Наскільки описані зміни вплинули на ефективність методу? Середня кількість порівнянь, хоч і зменшилася, але залишається O(n2), тоді як число обмінів не помінялося взагалі ніяк. Середня (вона ж гірша) кількість операцій залишається квадратичною.

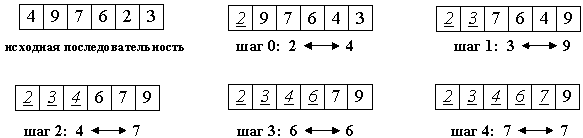
Додаткова пам'ять, очевидно, не потрібна. Поведінка вдосконаленого (але не початкового) методу досить природна, майже відсортований масив буде відсортований набагато швидше випадкового. Сортування бульбашкою стійке, проте шейкер-сортування втрачає цю якість.

**Сортування вибором**

Ідея методу полягає в тому, щоб створювати відсортовану послідовність шляхом приєднання до неї одного елементу за іншим в правильному порядку.

Будуватимемо готову послідовність, починаючи з лівого кінця масиву. Алгоритм складається з n послідовних кроків, починаючи від нульового і закінчуючи (n-1) -м.

На i-му кроці вибираємо найменший з елементів а[i]... а[n] і міняємо його місцями з а[i]. Послідовність кроків при n=5 зображена на малюнку нижче.



Незалежно від номера поточного кроку i, послідовність а[0]...a[i] (виділена курсивом) є впорядкованою. Таким чином, на (n-1) -му кроці вся послідовність, окрім а[n] виявляється відсортованою, а а[n] стоїть на останньому місці по праву: всі менші елементи вже пішли вліво.

Сортування вибором - простий алгоритм, на практиці не використовуваний зважаючи на низьку швидкодію. Натомість застосовують її поліпшення - пірамідальне сортування (Heapsort), а також (іноді) 'деревне сортування' (TreeSort).

{ Сортуються записи типу item по ключу item.key }

{ для допоміжних змінних використовується тип index }

procedure SelectSort;

var i, j, k: index; x:item;

begin for i:=1 to n-1 do

begin k:=i; x:=a[i];

for j:=i+1 to n do

if а[j].key < x.key then

begin k:=j; x:=a[j];

end;

а[k]:=a[i]; а[i]:=x;

end;

end;

Для знаходження найменшого елементу з n+1 елементу алгоритм здійснює n порівнянь. З урахуванням того, що кількість елементів, що розглядаються на черговому кроці, зменшується на одиницю, загальна кількість операцій:

n + (n-1) + (n-2) + (n-3) + ... + 1 = 1/2 \* ( n2+n ) = О(n2).

Таким чином, оскільки число обмінів завжди буде менше числа порівнянь, час сортування росте квадратично щодо кількості елементів.

Алгоритм не використовує додаткової пам'яті: всі операції відбуваються "на місці".

**Сортування вставками**

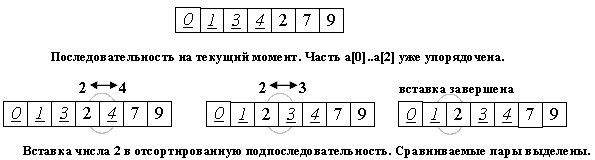
Сортування простими вставками в чомусь схоже на вищевикладені методи.

Аналогічним чином робляться проходи по частині масиву, і аналогічним же чином на його початку "зростає" відсортована послідовність...

Проте в сортуванні бульбашкою або вибором можна було чітко заявити, що на i-му кроці елементи а[0]...a[i] стоять на правильних місцях і нікуди більш не перемістяться. Тут же подібне твердження буде слабкішим: послідовність а[0]...a[i] впорядкована. При цьому по ходу алгоритму в неї вставлятимуться все нові елементи.

Розбиратимемо алгоритм, розглядаючи його дії на i-му кроці. Як мовилося вище, послідовність до цього моменту розділена на дві частини: готову а[0]...a[i] і неврегульовану а[i+1]...a[n].

На наступному, (i+1) -му кроці алгоритму беремо а[i+1] і вставляємо на потрібне місце в готову частину масиву. Пошук відповідного місця для чергового елементу вхідної послідовності здійснюється шляхом послідовних порівнянь з елементом, що стоїть перед ним. Залежно від результату порівняння елемент або залишається на поточному місці (вставка завершена), або вони міняються місцями і процес повторюється.



Таким чином, в процесі вставки ми "просіваємо" елемент x до початку масиву, зупиняючись у разі, коли

1. Hайдено елемент, менший x або
2. Досягнуто початку послідовності.

Аналогічно сортуванню вибором, середнє, а також гірше число порівнянь і пересилок оцінюються як О(n2), додаткова пам'ять при цьому не використовується.

Хорошим показником сортування є вельми природна поведінка: майже відсортований масив буде досортовано дуже швидко. Це, укупі із стійкістю алгоритму, робить метод хорошим вибором у відповідних ситуаціях.

**Порозрядне сортування**

Алгоритм, що розглядається нижче, істотно відрізняється від описаних раніше.

По-перше, він зовсім не використовує порівнянь сортованих елементів.

По-друге, ключ, по якому відбувається сортування, необхідно розділити на частини, *розряди* ключа. Наприклад, слово можна розділити по буквах, число - по цифрах...

До сортування необхідно знати два параметри: k і m, де

* k - кількість розрядів в найдовшому ключі
* m - розрядність даних: кількість можливих значень розряду ключа

При сортуванні російських слів m = 33, оскільки буква може приймати не більше 33 значень. Якщо в найдовшому слові 10 букв, k = 10.

Аналогічно, для для шістнадцяткових чисел m=16, якщо як розряд брати цифру, і m=256, якщо використовувати побайтове ділення.

Ці параметри не можна змінювати в процесі роботи алгоритму. У цьому - ще одна відмінність методу від вищеописаних.

Припустимо, що елементи лінійного списку L є к-розрядні десяткові числа, розрядність максимального числа відома наперед. Позначимо d(j,n) - j-у справа цифру числа n

Хай L0, L1,..., L9 - допоміжні списки (кишені), спочатку порожні. Порозрядне сортування складається з двох процесів, званих розподіл і збірка і виконуваних для j=1,2,...,k.

Фаза розподілу розносить елементи L по кишенях: елементи li списку L послідовно додаються в списки Lm, де m = d(j, li). Таким чином одержуємо десять списків, в кожному з яких j-ті розряди чисел однакові і рівні m.

Фаза збірки полягає в об'єднанні списків L0, L1,..., L9 у загальний список  
L = L0 => L1 => L2 => ... => L9

**Пірамідальне сортування**

Поступово переходимо від більш-менш простих до складних, але ефективних методів. Пірамідальне сортування одним з таких методів, швидкодія яких оцінюється як O(n log n).

Отже, назвемо пірамідою(Heap) бінарне дерево висоти k, в якому

* всі вузли мають глибину k або k-1 - дерево збалансоване.
* при цьому рівень k-1 повністю заповнений, а рівень k заповнений зліва направо, т.б форма піраміди має приблизно такий вигляд:  
  
* виконується "властивість піраміди": кожен елемент менше, або рівний батькові.

Як зберігати піраміду? Найпростіше - помістити її в масив.

|  |  |
| --- | --- |
|  | Відповідність між геометричною структурою піраміди як дерева і масивом встановлюється по наступній схемі:   * у а[0] зберігається корінь дерева * лівий і правий сини елементу а[i] зберігаються, відповідно, в а[2i+1] і а[2i+2] |

Таким чином, для масиву, що зберігає в собі піраміду, виконується наступна характеристична властивість: а[i]>= а[2i+1] і а[i]>= а[2i+2].

Плюси такого зберігання піраміди очевидні:

* ніяких додаткових змінних, потрібно лише розуміти схему.
* вузли зберігаються від вершини і далі вниз, рівень за рівнем.
* вузли одного рівня зберігаються в масиві зліва направо.

Відновити піраміду з масиву як геометричний об'єкт легко - досить пригадати схему зберігання і намалювати, починаючи від кореня.

***Фаза 1 сортування: побудова піраміди***

Почати побудову піраміди можна з а[k]...a[n], до = [size/2]. Ця частина масиву задовольняє властивості піраміди, оскільки не існує індексів i,j: i = 2i+1 ( або j = 2i+2 )... Просто тому, що такі i,j знаходяться за межею масиву.

Слід відмітити, що неправильно говорити про те, що а[k]..a[n] є пірамідою як самостійний масив. Це, взагалі кажучи, не вірно: його елементи можуть бути будь-якими. Властивість піраміди зберігається лише в рамках початкового, основного масиву а[0]...a[n].

Далі розширюватимемо частину масиву, що володіє такою корисною властивістю, додаючи по одному елементу за крок. Наступний елемент на кожному кроці додавання - той, який стоїть перед вже готовою частиною.

Щоб при додаванні елементу зберігалася пірамідальність, використовуватимемо наступну процедуру розширення піраміди а[i+1]..a[n] на елемент а[i] вліво:

1. Дивимося на синів зліва і справа - в масиві це а[2i+1] і а[2i+2] і вибираємо найбільшого з них.
2. Якщо цей елемент більше а[i] - міняємо його з а[i] місцями і йдемо до кроку 2, маючи на увазі нове положення а[i] в масиві. Інакше кінець процедури.

Новий елемент "просівається" крізь піраміду.

***Фаза 2: власне сортування***

Отже, завдання побудови піраміди з масиву успішно вирішене. Як видно з властивостей піраміди, в корені завжди знаходиться максимальний елемент. Звідси витікає алгоритм фази 2:

1. Беремо верхній елемент піраміди а[0]...a[n] (перший в масиві) і міняємо з останнім місцями. Тепер "забуваємо" про цей елемент і далі розглядаємо масив а[0]...a[n-1]. Для перетворення його в піраміду досить просіяти лише новий перший елемент.
2. Повторюємо крок 1, поки оброблювана частина масиву не зменшиться до одного елементу.

Очевидно, в кінець масиву кожного разу потрапляє максимальний елемент з поточної піраміди, тому в правій частині поступово виникає впорядкована послідовність.

*procedure Heapsort;*

*var i,l: index; x: item;*

*procedure sift;*

*label 13;*

*var i, j: index;*

*begin i := l; j := 2\*i; x := а[i];*

*while j <= r do*

*begin if j < r then*

*if а[j].key < а[j+1].key then j := j+1;*

*if x.key >= а[j].key then goto 13;*

*а[i]:= а[j]; i := j; j := 2\*i*

*end;*

*13: а[i]:= x*

*end;*

*begin l := (n div 2) + 1; r := n;*

*while l > 1 do*

*begin l := l - 1; sift*

*end;*

*while r > 1 do*

*begin x := а[l]; а[l]:= а[r]; а[r]:= x;*

*r := r - 1; sift*

*end*

*end { heapsort }*

Як швидкодія алгоритму, що вийшов ?

Побудова піраміди займає O(n log n) операцій, причому точніша оцінка дає навіть O(n) за рахунок того, що реальний час виконання downheap залежить від висоти вже створеної частини піраміди.

Друга фаза займає O(n log n) часу: O(n) раз береться максимум і відбувається просіювання елементу що був останнім. Плюсом є стабільність методу: середнє число пересилок (n log n) /2, і відхилення від цього значення порівняно малі.

Пірамідальне сортування не використовує додаткової пам'яті.

Метод не є стійким: по ходу роботи масив так "перетрушується", що початковий порядок елементів може змінитися випадковим чином.

Поведінка неприродна: часткова впорядкованість масиву ніяк не враховується.

# Швидке сортування

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

Ідея К. Хоара полягає в наступному:

1 Виберемо деякий елемент x масиву A випадковим чином;

2.Проглядаємо масив в прямому напрямі (i = 1, 2...), шукаючи в ньому елемент A[i] не менший, ніж x;

3.Проглядаємо масив у зворотному напрямі (j = n n-1,..), шукаючи в ньому елемент A[j] не більший, ніж x;

4.Міняємо місцями A[i] і A[j];

Пункти 2-4 повторюємо до тих пір, поки i < j;

В результаті такого стрічного 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 розділення ми витратимо не більше n/2 перестановок. Тепер залишилося виконати ті ж дії з початком і кінцем масиву, тобто застосувати їх рекурсивно.

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

*Procedure Swap(i, j : Integer);*

*Var b : Real;*

*Begin*

*b := а[i]; а[i]:= а[j]; а[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'єру x}*

*left := L; right := R ;*

*Repeat { стрічний прохід}*

*While A[left]< x do left := Succ(left); { перегляд вперед}*

*While A[right]> x do right := Pred(right); { перегляд назад}*

*If left <= right then begin*

*Swap(left, right); { перестановка}*

*left := Succ(left); right := Pred(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 виведення }*

*Begin*

*Inp; Hoare(1, n); Out*

*End.*

Найбільш тонке місце алгоритму Хоара - правильна обробка моменту закінчення руху покажчиків left, right. Відмітьте, що в нашій версії переставляються місцями елементи, рівні x. Якщо в циклах While замінити умови (A[left]< x) і (A[right]> x) на (A[left]<= x) і (A[right]>=x), при x = Max(A) індекси left і right пробіжать весь масив і побіжать далі. Ускладнення ж умов продовження перегляду погіршить ефективність програми.

Аналіз складності алгоритму в середньому, що використовує гіпотезу про рівну імовірність всіх входів, показує, що **C(n)= O(n log2 n), M(n)= O(n log2 n).**

У гіршому разі, коли як бар'єрний вибирається, наприклад, максимальний елемент підмасиву, складність алгоритму квадратична.

Поліпшити алгоритм швидкого сортування є великою спокусою: швидший алгоритм сортування - це своєрідна "мишоловка" для програмістів. Майже з тієї миті, як Хоар вперше опублікував свій алгоритм, в літературі стали з'являтися "покращувані" версії цього алгоритму. Було випробувано і проаналізовано безліч ідей, але все одно дуже просто помилитися, оскільки алгоритм настільки добре збалансований, що результатом поліпшення в одній його частині може стати сильніше погіршення в іншій його частині. Ми вивчимо в деяких деталях три модифікації цього алгоритму, які дають йому істотне поліпшення.

Добре відлагоджена версія швидкого сортування працюватиме набагато швидше, ніж будь-який інший алгоритм. Проте варто нагадати, що алгоритм дуже крихкий і будь-яка його зміна може привести до небажаних і несподіваних ефектів для деяких вхідних даних.

*program Quicksort;*

*uses*

*crt;*

*Const*

*N=10;*

*Type*

*Mas=array[1..n] of integer;*

*var*

*а: mas;*

*до: integer;*

*function Part(l, r: integer):integer;*

*var*

*v, i, j, b: integer;*

*begin*

*V:=a[r];*

*I:=l-1;*

*j:=r;*

*repeat*

*repeat*

*dec(j)*

*until (а[j]<=v) or (j=i+1);*

*repeat*

*inc(i)*

*until (а[i]>=v) or (i=j-1);*

*b:=a[i];*

*а[i]:=a[j];*

*а[j]:=b;*

*until i>=j;*

*а[j]:=a[i];*

*а[i]:= а[r];*

*а[r]:=b;*

*part:=i;*

*end;*

*procedure QuickSort(l, t: integer);*

*var i: integer;*

*begin*

*if l<t then*

*begin*

*i:=part(l, t);*

*QuickSort(l,i-1);*

*QuickSort(i+1,t);*

*end;*

*end;*

*begin*

*clrscr;*

*randomize;*

*for k:=1 to 10 do*

*begin*

*а[k]:=random(100);*

*write(а[k]:3);*

*end;*

*QuickSort(1,n);*

*writeln;*

*for k:=1 to n do*

*write(а[k]:3);*

*readln;*

*end.*

Методи поліпшення швидкого сортування.

## 1. Невеликі підфайли.

Перше поліпшення в алгоритмі швидкого сортування виникає із спостереження, що програма гарантовано викликає себе для величезної кількості невеликих підфайлів, тому слід використовувати найкращий метод сортування коли ми зустрічаємо невеликий підфайл. Очевидний спосіб добитися цього, це змінити перевірку на початку рекурсивної функції з "if r>l then" на виклик сортування вставкою (відповідно зміненої для сприйняття меж сортованого підфайлу): "if r-l<=M then insert(l, r)." Значення для M не зобов'язане бути "самим-самим" кращим: алгоритм працює приблизно однаково для M від 5 до 25. Час роботи програми при цьому знижується приблизно на 20% для більшості програм.

При невеликих підфайлах (5- 25 елементів) швидке сортування дуже багато раз викликає само себе (у прикладі для 10 елементів вона викликала сама себе 15 разів), тому слід застосовувати не швидке сортування, а сортування вставкою.

*procedure QuickSort (l,t:integer);*

*var*

*i:integer;*

*begin*

*if t-l>m then*

*begin*

*i:=part(l,t);*

*QuickSort (l,i-1);*

*QuickSort (i+1,t);*

*end*

*Else*

*Insert(l,t);*

*end;*

## 2. Ділення по медіані з трьох

Друге поліпшення в алгоритмі швидкого сортування полягає в спробі використання кращого ділячого елементу. У нас є декілька можливостей. Найбільш безпечною з них буде спроба уникнути гіршого випадку за допомогою вибору довільного елементу масиву як ділячого елементу. Тоді вірогідність гіршого випадку стає нехтівно малою. Це простий приклад "імовірнісного" алгоритму, який майже завжди працює незалежно від вхідних даних. Довільність може бути хорошим інструментом при розробці алгоритмів, особливо якщо можливі підозрілі вхідні дані.

Корисніше поліпшення полягає в тому, щоб узяти з файлу три елементи, і потім використовувати середнє з них як ділячий елемент. Якщо елементи узяті з початку, середини, і кінця файлу, то можна уникнути використання сторожових елементів: сортуємо узяті три елементи, потім обмінюємо центральний елемент з а[r-1], і потім використовуємо алгоритм ділення на масиві а[l+1..r-2]. Це поліпшення називається діленням по медіані з трьох.

Метод ділення по медіані з трьох корисний з трьох причин. По-перше, він робить вірогідність гіршого випадку набагато більш низкою. Щоб цей алгоритм використовував час пропорційний N2, два з трьох узятих елементів повинні бути або самими меншими, або найбільшими, і це повинно повторюватися з розділу в розділ. По-друге, цей метод знищує необхідність в сторожових елементах, оскільки цю роль грає один з трьох узятих нами перед діленням елементів. У третіх, він насправді знижує час роботи алгоритму приблизно на 5%.

*procedure exchange(i,j:integer);*

*var*

*k:integer;*

*begin*

*k:=a[i];*

*а[i]:=a[j];*

*а[j]:=k;*

*end;*

*procedure Mediana;*

*var i:integer;*

*begin*

*i:=n div 4;{Мал.}*

*if а[i]>a[i\*2] then*

*if а[i]>a[i\*3] then*

*exchange(i,n)*

*else*

*exchange(i\*3,n)*

*else*

*if а[i\*2]>a[i\*3] then*

*exchange(i\*2,n);*

*quicksort(1,n);*

*end;*

## 3. Нерекурсивна реалізація.

Можна переписати даний алгоритм без використання рекурсії використовуючи стек.

Комбінація нерекурсивної реалізації ділення по медіані з трьох з відсіканням на невеликі файли може поліпшити час роботи алгоритму від 25% до 30%.

## Злиття

Зовнішнє сортування сортує файли, які не поміщаються цілком в оперативну пам'ять.

Зовнішнє сортування сильно відрізняється від внутрішнього. Річ у тому, що доступ до файлу є послідовним, а не паралельним як це було в масиві. І тому прочитувати файл можна тільки блоками і цей блок відсортувати в пам'яті і знову записати у файл.

Принципову можливість ефективно відсортувати файл, працюючи з його частинами і не виходячи за межі частини забезпечує алгоритм злиття.

Під злиттям розуміється об'єднання двох (або більш) впорядкованих послідовностей в одну впорядковану послідовність за допомогою циклічного вибору елементів, доступних в даний момент.

Злиття - набагато простіша операція, ніж сортування.

Ми розглянемо 2 алгоритми злиття:

* Пряме злиття. Алгоритм Боуза - Нельсона
* Природне (Нейманівське) злиття.

## Пряме злиття. Алгоритм Боуза - Нельсона

Послідовність а розбивається на дві половини b і с.

Послідовності b і с зливаються за допомогою об'єднання окремих елементів у впорядковані пари.

Одержаній послідовності привласнюється ім'я а, після чого повторюються кроки 1 і 2; при цьому впорядковані пари зливаються у впорядковані четвірки.

Попередні кроки повторюються, при цьому четвірки зливаються у вісімки і т.д., поки не буде впорядкована вся послідовність, оскільки довжини послідовностей кожного разу подвоюються.

Головним мінусом сортування злиттям є подвоєння розміру пам'яті, спочатку зайнятої сортованими даними. Розглянемо алгоритм з рекурсивним актом злиття, запропонований Боузом і Нельсоном і що не вимагає резерву пам'яті.

Він заснований на очевидній ідеї: злити дві рівні впорядковані частини можна злиттям їх початкових половин, злиттям кінцевих і злиттям 2-ї половини 1-го результату з 1-ю половиною 2-го результату, наприклад:

Якщо частини не рівні або не діляться точно навпіл, процедуру уточнюють належним чином. Аналогічно злиття "половинок" можна звести до злиття "четвертинок", "восьмушок" і т. д.; має місце рекурсія.

*Const n=200;*

*Type*

*tipkl=word;*

*tip = Record*

*kl: tipkl;*

*z:Array[1..50] of real*

*End;*

*Var*

*A: Array[1..n] of tip;*

*j:word;*

*Procedure Bose (Var AA; voz:Boolean);*

*Var*

*m,j:word; x:tip; {tip - тип сортованих записів}*

*A: Array [1..65520 div Sizeof(tip)] of tip Absolute AA;*

*Procedure Sli(j,r,m: word); { r - відстань між початками*

*зливаних частин, а m - їх розмір, j - найменший номер запису}*

*Begin*

*if j+r<=n Then*

*If m=1 Then*

*Begin*

*If voz Xor (A[j].kl < A[j+r].kl) Then*

*Begin*

*x:=A[j];*

*A[j]:= A[j+r];*

*A[j+r]:=x*

*End*

*End*

*Else*

*Begin*

*m:=m div 2;*

*Sli(j,r,m); { Злиття "почав"}*

*If j+r+m<=n Then*

*Sli(j+m,r,m); { Злиття "кінців"}*

*Sli(j+m,r-m,m) End { Злиття в центральній частині}*

*End{блоку Sli};*

*Begin*

*m:=1;*

*Repeat*

*j:=1; { Цикл злиття списків рівного розміру: }*

*While j+m<=n do*

*Begin*

*Sli(j,m,m);*

*j:=j+m+m*

*End;*

*m:=m+m { Подвоєння розміру списку перед початком нового проходу}*

*Until m >= n { Кінець циклу, що реалізовує все дерево злиття}*

*End{блоку Bose};*

*BEGIN*

*Randomize;*

*For j:=1 to n do*

*begin*

*A[j].kl:= Random(65535);*

*Write(A[j].kl:8);*

*end;*

*Readln;*

*Bose(A,true);*

*For j:=1 to n do*

*Write(A[j].kl:8);*

*Readln*

*END.*

## Природне (Нейманівське) злиття.

Воно об'єднує впорядковані частини, що спонтанно виникли в початковому масиві; вони можуть бути також наслідком попередньої обробки даних. Розраховувати на однаковий розмір зливаних частин не доводиться.

Записи, що йдуть у порядку неубування ключів, зчіплюються, утворюючи підсписок. Мінімальний підсписок один запис.

Для програмної реалізації заводять масив sp: елемент sp[i] - це номер запису, який слідує за i-м.

Останній запис одного підсписку посилається на перший запис іншого, а для розрізнення кінців підсписків це посилання забезпечується знаком мінус.

*Repeat { Повторення актів злиття підсписків}*

*If A[j].kl < A[i].kl Then { Вибирається менший запис}*

*Begin sp[k]:=j; k:=j; j:=sp[j];*

*If j<=0 Then { Зчеплення із залишком "i"-подсписка}*

*Begin sp[k]:=i; Repeat m:=i; i:=sp[i] Until i<=0 End*

*End*

*Else*

*Begin sp[k]:=i; k:=i; i:=sp[i];*

*If i<=0 Then { Зчеплення із залишком "j"-подсписка}*

*Begin sp[k]:=j; Repeat m:=j; j:=sp[j] Until j<=0 End*

*End;*

*If j<=0 Then Begin sp[m]:= 0; sp[p]:=-sp[p]; i:=-i;*

*j:=-j; If j<>0 Then p:=r; k:=r; r:=m End*

*Until j=0;*

{ У кінець сформованого підсписку завжди заноситься нульове посилання (sp[m]:= 0), бо він може виявитися останнім.

Дія sp[p]:= -sp[p] позначає мінусом кінець раніше побудованого підсписку.

У змінних i,j посилання на початки нових зливаних підсписків - із знаком мінус; його знімаємо. Перехід до нових підсписків вимагає оновлення змінних p, k, r}

**Сортування двійковим деревом, 'деревне сортування'.**

Двійковим (бінарним) деревом назвемо впорядковану структуру даних у якій кожному елементу - попереднику або кореню (під) дерева - поставлені у відповідність принаймні два інші елементи (наступника).

Причому для кожного попередника виконано наступне правило: лівий наступник завжди менше, а правий наступник завжди більше або рівний попереднику.

При додаванні в дерево нового елементу його послідовно порівнюють з нижчестоячими вузлами, таким чином вставляючи на місце.

Якщо елемент >= кореня - він йде в праве піддерево, порівнюємо його вже з правим сином, інакше - він йде в ліве піддерево, порівнюємо з лівим і так далі, поки є сини, з якими можна порівняти.

Якщо ми рекурсивно обходитимемо дерево за правилом "лівий син - батько - правий син", то, записуючи всі елементи, що зустрічаються, в масив, ми одержимо впорядковану у порядку зростання множину. Hа цьому і заснована ідея сортування деревом.

Загальна швидкодія методу O(nlogn). Поведінка неприродна, стійкості, взагалі кажучи, немає.

Основний недолік цього методу - великі вимоги до пам'яті під дерево. Очевидно, потрібно n місця під ключі і, крім того, пам'ять на 2 покажчики для кожного з них.

Тому TreeSort звичайно застосовують там, де

- побудоване дерево можна з успіхом застосувати для інших завдань.

- дані вже побудовані в 'дерево'

- дані можна прочитувати безпосередньо в дерево.

Hаприклад, при потоковому введенні з консолі або з файлу.

**Багатофазне сортування**

Просте сортування злиттям цілком можна застосувати до будь-якого файлу і воно працюватиме.

Розберемо серйозно покращений алгоритм зовнішнього сортування злиттям що працює набагато швидше.

***Багатофазне сортування злиттям.***

Злиттям називається процес об'єднання декількох впорядкованих серій (т.б впорядкованих списків) в одну.

Hа кожному кроці ми беремо найменший з початкових елементів вхідних серій і переміщаємо в кінець вихідної серії. Кожна операція злиття серій, очевидно, вимагає n пересилок елементів, де n - загальне число елементів серій.

В процесі сортування ми оперуватимемо стрічками - структурами даних, де в кожен момент нам доступний або перший елемент, або наступний елемент після вже прочитаного. У реальній ситуації як стрічки виступають однозв'язні списки або файли.

Хай у нас є N стрічок: N - 1 вхідна і одна порожня. Ми будемо зливати елементи з вхідних стрічок на вихідну, поки яка-небудь з них не спустіє. Потім вона стане вхідною.

***Завдання:***

При виконанні завдання потрібно:

- сортувати множину, що складається не менш ніж з 100 елементів;

- оцінити складність виконаного алгоритму;

- скласти блок-схему результуючої програми.

**Варіант 1**

Реалізувати алгоритм TreeSort, застосувавши метод вкладення дерева в масив.

**Варіант 2**

Реалізувати ітеративну версію алгоритму пірамідального сортування.

**Варіант 3**

Реалізувати ітеративну версію алгоритму Хоара.

**Варіант 4**

Реалізувати алгоритм шейкерного сортування.

**Варіант 5**

Реалізувати алгоритм сортування вибором.

**Варіант 6**

Реалізувати алгоритм сортування вставками.

**Варіант 7**

Реалізувати рекурсивну версію алгоритму пірамідального сортування.

**Варіант 8**

Реалізувати рекурсивну версію алгоритму Хоара.

**Варіант 9**

Реалізувати модифікований алгоритм швидкого сортування з врахуванням сортування невеликих підмножин.

**Варіант 10**

Реалізувати модифікований алгоритм швидкого сортування з використанням ділення по медіані з трьох.

**Варіант 11**

Реалізувати алгоритм сортування прямим злиттям.

**Варіант 12**

Реалізувати алгоритм сортування природнім злиттям.

**Варіант 13**

Реалізувати ітеративний алгоритм швидкого сортування.

**Варіант 14**

Реалізувати алгоритм бульбашкового сортування.

**Варіант 15**

Реалізувати алгоритм порозрядного сортування

***Контрольні питання:***

1. Що таке зовнішнє сортування?
2. Що таке внутрішнє сорування?
3. Суть методу бульбашкового сортування?
4. В чому полягає особливість алгоритму шейкерного сортування?
5. Алгоритм швидкого сортування Хоара?
6. Суть сортування злиттям?
7. Властивості алгоритмів сортування?