МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет "Львівська політехніка"

Алгоритми сортування

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторної роботи № 10
з курсу **"Теорія алгоритмів і структури даних"**
для студентів базового напрямку 6.0804
"Комп'ютерні науки"

Затверджено на засіданні кафедри
"Системи автоматизованого проектування"
Протокол N 1 від 31.08.2006р.

Львів 2006

Алгоритми сортування. Методичні вказівки до лабораторної роботи №10 з курсу “Теорія алгоритмів і структури даних” для студентів базового напрямку 6.0804 - "Комп'ютерні науки" / Укл. А.Б.Керницький - Львів: НУ “ЛП”, 2006. - ---с.

Укладач: А.Б.Керницький, др., ст.викл.

Відповідальний за випуск С.П.Ткаченко, канд.техн.наук, доц.

Рецензенти: Ю.В.Стех, канд.техн.наук, доц.
І.І.Мотика, канд.техн.наук, доц.

1. МЕТА РОБОТИ

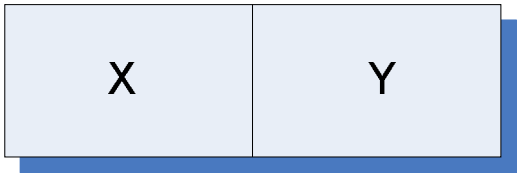
Метою роботи є вивчення алгоритмів сортування.

2. КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Сортуванням називають впорядковування по ключах (тобто за якою-небудь ознакою) елементів деякої структури даних на якій визначено відношення порядку. У залежності від того, чи знаходяться елементи структур даних у внутрішній (оперативній) пам'яті або у зовнішній пам'яті (на зовнішніх пристроях), розрізняють внутрішнє та зовнішнє сортування.

Нехай існують послідовність a_0, a_1... a_n і функція порівняння, яка на будь-яких двох елементах послідовності приймає одне з трьох значень: менше, більше або дорівнює. Задача сортування полягає у перестановці елементів послідовності так, щоб виконувалася умова: $a_i <= a_{i+1}$, для всіх i від 0 до n.

Можлива ситуація, коли елементи складаються з декількох полів (рис.4):




```
struct element {
  field x;
  field y;
  }
```

Рис.4 Структура елемента послідовності

Якщо значення функції порівняння залежить тільки від поля x, то x називають ключем, по якому проводиться сортування. На практиці, в якості x часто виступає число, а поле y зберігає будь-які дані, жодним чином не впливаючи на роботу алгоритму.

Мабуть, жодна інша проблема не породила такої кількості найрізноманітніших рішень, як задача сортування. Чи існує певний "універсальний, найкращий алгоритм"? Маючи приблизні характеристики вхідних даних, можна підібрати метод, який працює оптимальним чином.

Для того, щоб обґрунтовано зробити такий вибір, розглянемо параметри, по яких проводититься оцінка алгоритмів.

1. *Час* сортування - основний параметр, що характеризує швидкодію алгоритму.

2. *Пам'ять* - ряд алгоритмів вимагає виділення додаткової пам'яті під тимчасове зберігання даних. При оцінці пам'яті, що використовується, не враховується місце, яке займає початковий масив і незалежні від вхідної послідовності витрати, наприклад, на зберігання коду програми.

3. *Стійкість* - стійке сортування не міняє взаємного розташування рівних елементів. Така властивість може бути дуже корисною, якщо вони складаються з декількох полів, як на рис. 5, а сортування відбувається по одному з них, наприклад, по x.

Рис.5 Приклад стійкого і нестійкого сортування

Взаємне розташування рівних елементів з ключем 1 і додатковими полями "a", "b", "c" залишилося початковим: елемент з полем "a", потім - з "b", потім - з "c".

Взаємне розташування рівних елементів з ключем 1 і додатковими полями "a", "b", "c" змінилося.

4. *Природність поведінки* - ефективність методу при обробці вже відсортованих, або частково відсортованих даних. Алгоритм поводиться природно, якщо враховує цю характеристику вхідної послідовності і працює краще.

Ще однією важливою властивістю алгоритму є його сфера застосування. Тут основних позицій дві:

- внутрішні сортування працюють з даним в оперативній пам'яті з довільним доступом;
- зовнішні сортування упорядковують інформацію, розташовану на зовнішніх носіях. Це накладає деякі додаткові обмеження на алгоритм:
  - доступ до носія здійснюється послідовним чином: в кожний момент часу можна зчитати або записати тільки елемент, наступний за юіжучим;
  - об’єм даних не дозволяє їм розміститися в ОПЗ;

Крім того, доступ до даних на носію здійснюється набагато повільніше, ніж операції з оперативною пам’яттю.

Даний клас алгоритмів ділиться на два основні підкласи:

- *Внутрішнє сортування* оперує з масивами, що цілком поміщаються в оперативній пам’яті з довільним доступом до будь-якої комірки. Дані звичайно сортуються на тому ж місці, без додаткових витрат.
- *Зовнішнє сортування* оперує з пристроями великого об'єму, що запам’ятовують, але з доступом не довільним, а послідовним (сортування файлів), тобто у даний момент ми “бачимо” тільки один елемент, а витрати на перемотування у порівнянні з пам’яттю невиправдано великі . Це приводить до спеціальних методів сортування, які звичайно використовують додатковий дисковий простір.

1.1 Сортування вибором

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

Будуватимемо відсортовану 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 a[i]...a[n] і міняємо його місцями з a[i]. Послідовність кроків при n=5 зображена на рисунку нижче.

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

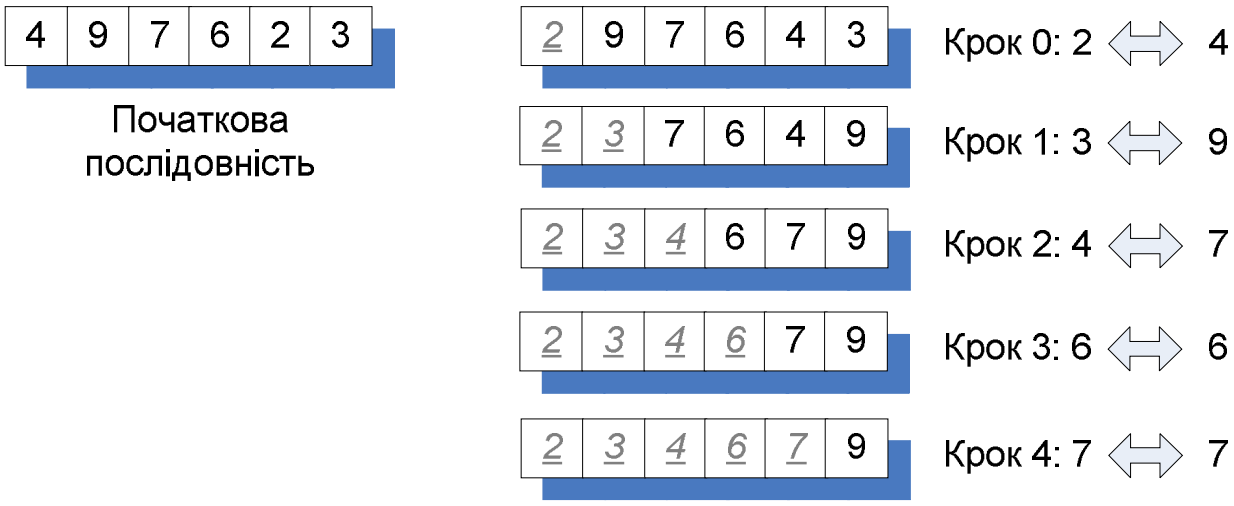


Рис.6 Приклад роботи алгоритму сортування вибором

Для знаходження найменшого елемента з n+1, які розглядаються, алгоритм виконує n порівнянь. Із врахуванням того, що кількість елементів, які розглядаються, на черговому кроці зменшується на одиницю, загальна кількість операцій:

$n + (n-1) + (n-2) + (n-3) + ... + 1 = 1/2 * (n^2+n) = Theta(n^2)$.

Таким чином, оскільки кількість обмінів завжди буде менша кількості порівнянь, час сортування росте квадратично відносно кількості елементів.

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

Розглянемо послідовність з трьох елементів, кожний з яких має два поля, а сортування йде по першому з них.

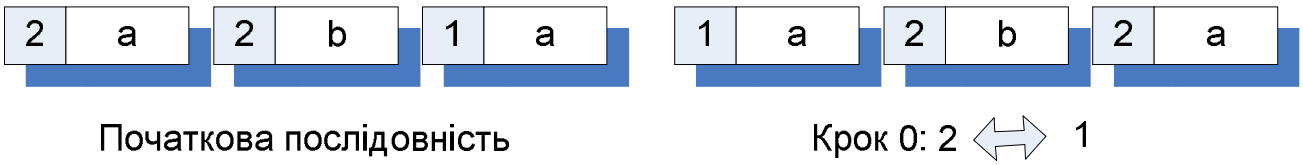


Рис.7 Алгоритм сортування вибором (елементи мають 2 поля)

Результат її сортування можна побачити вже після кроку 0, оскільки більше обмінів не буде. Порядок ключів 2a, 2b був змінений на 2b, 2a, так що метод нестійкий.

Якщо вхідна послідовність майже впорядкована, то порівнянь буде стільки ж, значить алгоритм поводиться неприродно.

1.2 Бульбашкове сортування

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

Ідея методу: крок сортування полягає в проході знизу вверх по масиву. У процесі просування по масиве порівнюємо пари сусідніх елементів. Якщо елементи деякої пари знаходяться в неправильному порядку, то міняємо їх місцями.

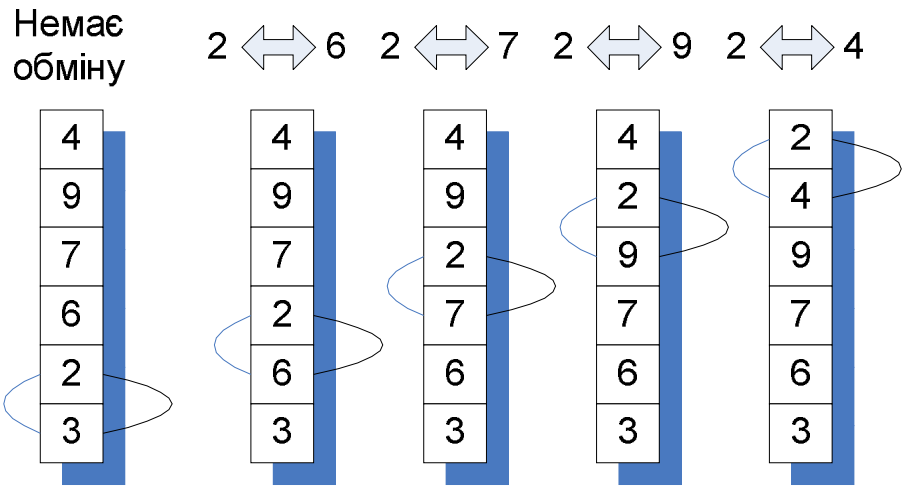


Рис.8 Нульовий прохід у бульбашковому сортування; порівнювані пари виділені

Після нульового проходу по масиву (рис.8) "вгорі" виявляється "найлегший елемент" - звідси аналогія з бульбашкою. Наступний прохід робиться до другого зверху елемента, таким чином другий за величиною елемент підіймається на правильну позицію.

Робимо проходи по нижній частині масиву, який із кожним проходом зменшується, доти, поки в ній залишиться тільки один елемент. На цьому сортування закінчується, оскільки послідовність впорядкована за зростанням.

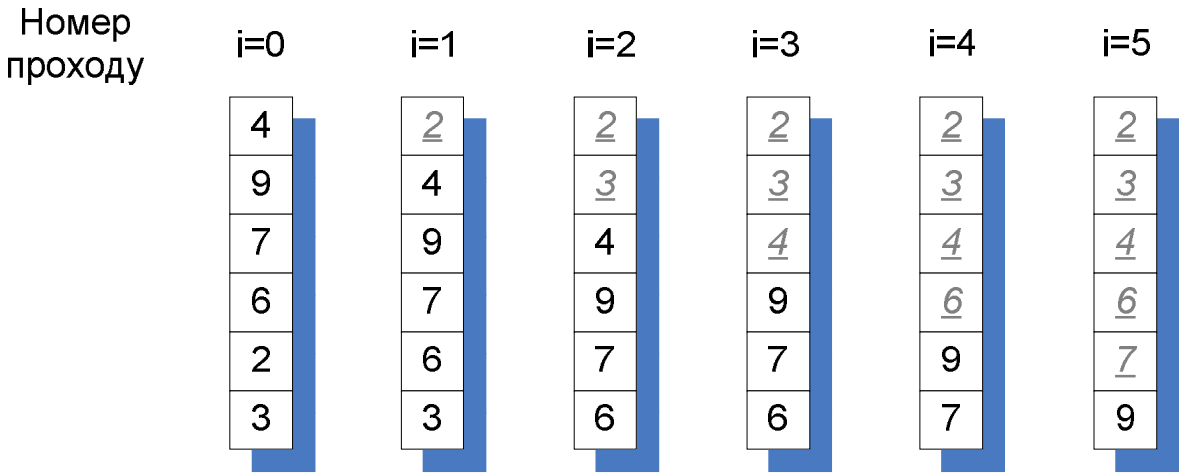


Рис.9 Приклад роботи алгоритму бульбашкового сортування

Середня кількість порівнянь і обмінів мають квадратичний порядок зростання: Theta(n^2), звідси можна зробити висновок, що бульбашковий алгоритм дуже повільний і малоефективний.

Проте, у даного алгоритму є великий плюс: він простий і його можна покращувати. По-перше, розглянемо ситуацію, коли на якому-небудь з проходів не відбулося жодного обміну. Це означає, що всі пари розташовані у правильному порядку, так що масив вже відсортований, і продовжувати процес не має сенсу.

Отже, перше покращання алгоритму полягає у запам'ятовуванні, чи проводився на даному проході який-небудь обмін. Якщо ні - алгоритм закінчує роботу.

Процес покращення можна продовжити, якщо запам'ятовувати не лише сам факт обміну, але і індекс останнього обміну k. Дійсно: всі пари сусідніх елементів з індексами, меншими k, вже розташовані у потрібному порядку. Подальші проходи можна закінчувати на індексі k, замість того щоб рухатися до встановленої наперед верхньої межі i.

Якісно інше покращення алгоритму можна отримати 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 напрямок проходів, які ідуть один за одним. Отриманий алгоритм іноді називають "*шейкер-сортуванням*".

Наскільки описані зміни вплинули на ефективність методу? Середня кількість порівнянь, хоча і зменшилося, але залишається O(n^2), тоді як кількість обмінів не змінилась взагалі. Середня кількість операцій залишається квадратичною.

Додаткова пам'ять не потрібна. Поведінка вдосконаленого (але не початкового) методу досить природна, майже відсортований масив буде

відсортований набагато швидше випадкового. Бульбашкове сортування є стійким, проте шейкер-сортування втрачає цю якість.

На практиці бульбашковий метод, навіть із вдосконаленнями, працює, на жаль, дуже повільно. А тому - майже не застосовується.

1.3 Сортування вставками

Алогоритм сортування простими вставками у чомусь подібний на вищенаведені два методи.

У ньому аналогічним чином робляться проходи по частині масиву, і аналогічним чином у його початку "зростає" відсортована послідовність.

Проте у бульбашковому сортуванні або сортуванні вибором можна було чітко заявити, що на i-му кроці елементи a[0]...a[i] стоять на правильних місцях і більше нікуди не перемістяться. У даному алгоритмі подібне твердження буде більш слабшим: послідовність a[0]...a[i] впорядкована. При цьому у процесі функціонування алгоритму в неї вставлятимуться (див. назву методу) все нові елементи.

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

На наступному, (i+1)-у кроці алгоритму беремо a[i+1] і вставляємо на потрібне місце в готову частину масиву.

Пошук відповідного місця для чергового елемента 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.

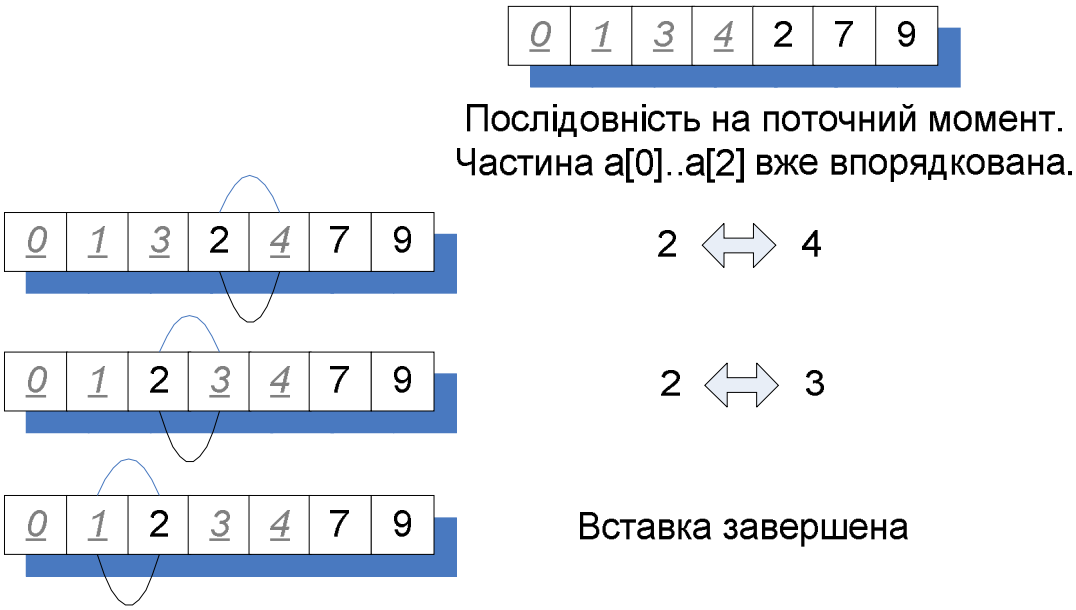


Рис.10 Алгоритм сортування вставками. Вставка числа 2 в послідовність. Порівнювані пари виділені

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

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

Аналогічно сортуванню вибором, середня, а також найгірша кількість порівнянь і пересилок оцінюються як Theta(n^2), додаткова пам’ять при цьому не використовується.

Добрим показником сортування є досить природна поведінка: майже відсортований масив буде досортований дуже швидко. Це, разом із стійкістю алгоритму, робить метод хорошим вибором у відповідних ситуаціях.

Даний алгоритм можна дещо покращити. На кожному кроці внутрішнього циклу перевіряються 2 умови. Можна об’єднати їх в одну, поставивши у початок

масиву спеціальний вартовий елемент. Він повинен бути наперед меншим за всю решту елементів масиву.

Рис.11 Використання спеціального вартового елементу у покращеному алгоритмі сортування вставками

Тоді при j=0 буде заздалегідь вірно a[0]<=x. Цикл зупиниться на нульовому елементі, що і було метою умови j>=0.

Таким чином, сортування відбуватиметься правильним чином, а у внутрішньому циклі стане на одне порівняння менше. Із врахуванням того, що воно проводилося Theta(n^2) раз, це - реальна перевага. Проте, відсортований масив не буде повний, оскільки з нього зникло перше число. Для закінчення сортування це число слід повернути назад, а потім вставити у відсортовану послідовність a[1]...a[n].

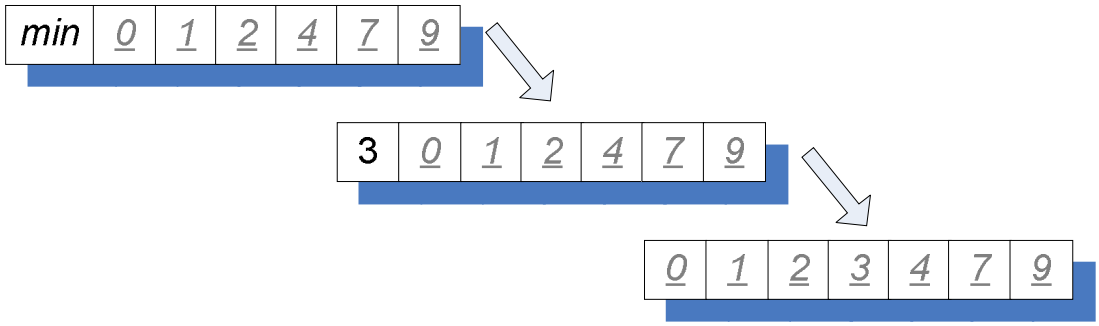
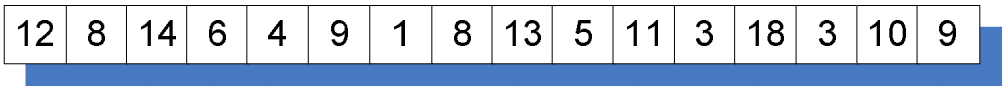


Рис.12 Приклад повернення усунутого числа у покращеному алгоритмі сортування вставками

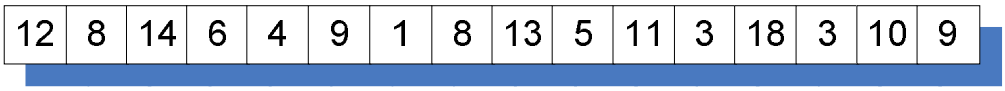
1.4 Сортування Шелла

Сортування Шелла є досить цікавою модифікацією алгоритму сортування простими вставками.

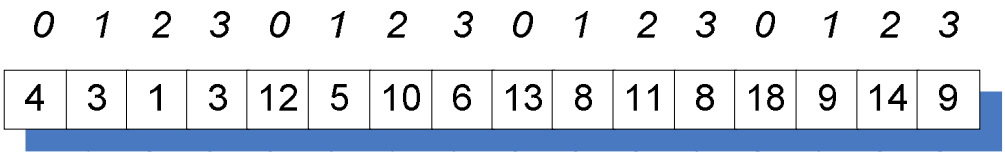
Розглянемо наступний алгоритм сортування масиву a[0].. a[15].



1. Спочатку сортуємо простими вставками кожні 8 груп з 2-х елементів (a[0], a[8[), (a[1], a[9]) ... (a[7], a[15]).

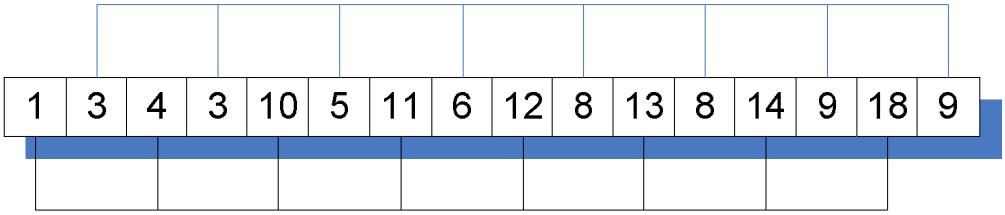


2. Потім сортуємо кожну з чотирьох груп по 4 елементи (a[0], a[4], a[8], a[12]) ... (a[3], a[7], a[11], a[15]).

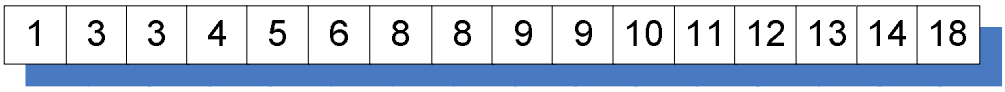


У нульовій групі будуть елементи 4, 12, 13, 18, у першій - 3, 5, 8, 9 і т.д.

3. Далі сортуємо 2 групи по 8 елементів, починаючи з (a[0], a[2], a[4], a[6], a[8], a[10], a[12], a[14]).



4. У кінці сортуємо вставками всі 16 елементів.



Очевидно, лише останнє сортування є необхідним, щоб розташувати всі елементи по своїх місцях. Решта сортувань просувають елементи максимально близько до відповідних позицій, так що на останній стадії кількість переміщень буде не дуже велика. Послідовність і так майже відсортована. Прискорення

підтверджене численними дослідженнями і на практиці виявляється досить істотним.

Єдиною характеристикою сортування Шелла є *приріст* - відстань між сортованими елементами, в залежності від проходу. У кінці приріст завжди дорівнює одиниці - метод завершується звичайним сортуванням вставками, але саме послідовність приростів визначає зростання ефективності.

Використаний у прикладі набір ..., 8, 4, 2, 1 - непоганий вибір, особливо, коли кількість елементів - степінь двійки. Проте набагато кращий варіант запропонував Р.Седжвік [19]. Його послідовність має вигляд:

$$inc[s] = \begin{cases} 9*2^s - 9*2^{s/2} + 1, якщо - s - парне \\ 8*2^s - 6*2^{(s+1)/2} + 1, якщо - s - непарне \end{cases}$$

При використанні таких приростів середня кількість операцій: $O(n^{7/6})$, у гіршому випадку - порядку $O(n^{4/3})$.

Звернемо увагу на те, що послідовність обчислюється у порядку, протилежному тому, що використовується: inc[0]= 1, inc[1]= 5 ... Формула дає спочатку менші числа, потім все більші і більші, тоді як відстань між сортованими елементами, навпаки, повинна зменшуватися. Тому масив приростів inc обчислюється перед запуском власне сортування до максимальної відстані між елементами, яке буде першим кроком у сортуванні Шелла. Потім його значення використовуються в зворотному порядку.

При використанні формули Седжвіка слід зупинитися на значенні inc[s-1], якщо 3*inc[s]> size.

Часто замість обчислення послідовності під час кожного запуску процедури, її значення розраховують наперед і записують у таблицю, якою користуються, вибираючи початковий приріст за тим же правилом: починаємо з inc[s-1], якщо 3*inc[s]> size.

1.5 Сортування злиттям

Спочатку розглянемо задачу злиття двох впорядкованих масивів *a* та *b* з *m* та *n* елементів у впорядкований масив c з (*m*+*n*) елементів, що вміщує всі елементи масивів *a* та *b*. Цю задачу розв'язує фрагмент алгоритма:

```
i <- 1; j <- 1; k <- 1;
поки i<=n & j<=m повт
  якщо a[i]<b[j] то
    c[k] <- a[i]; i <- i+1
  інакше
    c[k] <- b[j]; j <- j+1
  кр;
  k <- k+1
кц;
поки i<=n повт
  c[k] <- a[i]; i <- i+1;
  k <- k+1
кц;
поки j<=m повт
  c[k] <- b[j]; j <- j+1;
  k <- k+1
кц;
```

Для кожного конкретного випадку другий або третій цикл не виконується жодного разу у залежності від того, який масив вичерпується раніше: *a* або *b*. Зазначимо, що злиття вимагає O(n+m) операцій.

При сортуванні злиттям на i-му кроці масив *a* розбивається на впорядковані підмасиви (послідовності елементів масиву, що йдуть підряд) довжини Size=2i-1. Пари сусідніх підмасивів зливаються у впорядковані підмасиви довжини Size*2 у допоміжному масиві *b*. Після присвоєння *a* значення *b* та збільшення вдвічі значення Size злиття повторюється для підмасивів більшого розміру. Процес завершується, коли Size стає більшим або рівним *n*. Оскільки при i=1 підмасиви з 1

елемента впорядковані за означенням, у результаті отримуємо впорядкований масив *a*.

На високому рівні алгоритм можна представити так:

```
Size <- 1;
повт
        злити пари впорядкованих підмасивів довжини Size з a у b;
        a <- b;
        Size <- Size*2;
до Size > n;
```

Для злиття пар впорядкованих підмасивів довжини Size з a у b визначимо процедуру Merge. В ній r1 – права границя індексів першого підмасиву пари, r2 – права границя індексів другого підмасиву пари. Оскільки Size приймає значення степені 2, а n може не бути кратним степені 2, довжина підмасивів останньої пари може бути менше Size.

```
процедура Merge(арг a: τ; Size: нат; рез b: τ) це
        змін i, j, k, r1, r2: нат;
поч
        k <- 1;
        поки k<=n повт
          {визначення границь підмасивів}
          i <- k; r1 <- i+Size-1;
          якщо r1>n то r1 <- n кр;
          j <- r1+1; r2 <- j+Size-1;
          якщо r2>n то r2 <- n кр;
          {злиття пари підмасивів}
          поки i<=r1 & j<=r2 повт
                якщо a[i]<a[j] то
                  b[k] <- a[i]; i <- i+1
                інакше
```

```
                b[k] <- a[j]; j <- j+1
            кр;
            k <- k+1
        кц;
        поки i<=r1 повт
            b[k] <- a[i]; i <- i+1;
            k <- k+1
        кц;
        поки j<=r2 повт
            b[k] <- a[j]; j <- j+1;
            k <- k+1
        кц;
    кц;
кп;
```

Перепризначення a <- b можна зробити тільки один раз, якщо на непарних кроках циклу **повт** ... **до** виконувати злиття з a у b, а на парних - з b в a.

Остаточно отримуємо:

```
процедура MergeSort (мод a: τ) це
    процедура Merge(арг a: τ; Size: нат; рез b: τ) це
        змін i, j, k, r1, r2: нат;
    поч
        k <- 1;
        поки k<=n повт
          {визначення границь підмасивів}
          i <- k; r1 <- i+Size-1;
          якщо r1>n то r1 <- n кр;
          j <- r1+1; r2 <- j+Size-1;
          якщо r2>n то r2 <- n кр;
          {злиття пари підмасивів}
          поки i<=r1 & j<=r2 повт
              якщо a[i]<a[j] то
```

```
                    b[k] <- a[i]; i <- i+1
                  інакше
                    b[k] <- a[j]; j <- j+1
                  кр;
                  k <- k+1
              кц;
              поки i<=r1 повт
                  b[k] <- a[i]; i <- i+1;
                  k <- k+1
              кц;
              поки j<=r2 повт
                  b[k] <- a[j]; j <- j+1;
                  k <- k+1
              кц;
            кц;
        кп;
        змін i, Size: нат; b: τ;
    поч
      i <- 0; Size <- 1;
                повт
        i <- i+1
        якщо i mod 2 = 1 то Merge(a, Size, b)
        інакше Merge(b, Size, a) кр;
        Size <- Size*2;
      до Size >= n;
      якщо i mod 2 = 1 то a <- b кр
    кп;
```

Послідовність сортування масива по кроках:

```
    23 55 47 35 10 90 84 30
i=1  23 55 35 47 10 90 30 84
i=2  23 35 47 55 10 30 84 90
```

i=3 10 23 30 35 47 55 84 90

Якщо розглянути необхідну кількість дій, то вона має порядок $O(n\log_2 n)$. Дійсно, цикл **повт** ... **до** виконується $\log_2 n$ раз, а на кожному його кроці виконання злиття вимагає O(n) операцій. Отже, сортування злиттям є більш швидким у порівнянні з розглянутими раніше методами, але вимагає додаткової пам'яті порядку n.

1.6 Пірамідальне сортування

Пірамідальне сортування є першим з ефективних методів, швидкодія яких оцінюється як O(n log n).

Розглянемо перевернуте сортування вибором. Під час проходу, замість вставки найменшого елемента у лівий кінець масиву, вибиратимемо найбільший елемент, а готову послідовність будемо будувати у правому кінці.

Приклад дій для масиву a[0]... a[7]:

44 55 12 42 94 18 06 67	початковий масив
44 55 12 42 67 18 06 |94	94 <-> 67
44 55 12 42 06 18 |67 94	67 <-> 06
44 18 12 42 06 |55 67 94	55 <-> 18
06 18 12 42 |44 55 67 94	44 <-> 06
06 18 12 |42 44 55 67 94	42 <-> 42
06 12 |18 42 44 55 67 94	18 <-> 12
06 |12 18 42 44 55 67 94	12 <-> 12

Вертикальною межею визначено ліву межу вже відсортованої (правої) частини масиву.

Розглянемо оцінку кількості операцій докладніше. Всього виконується n кроків, кожний з яких полягає у виборі найбільшого елемента з послідовності

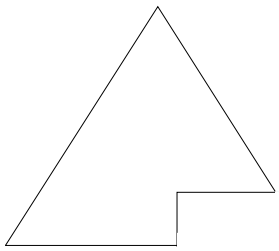
a[0]..a[i] і подальшому обміні. Вибір відбувається послідовним перебором елементів послідовності, тому необхідний на нього час: O(n). Отже, n кроків по O(n) кожний - це $O(n^2)$.

Проведемо вдосконалення: побудуємо структуру даних, що дозволяє вибирати максимальний елемент послідовності не за O(n), а за O(log n) час. Тоді загальна швидкодія сортування буде n*O(log n) = O(n log n).

Ця структура також повинна дозволяти швидко вставляти нові елементи (щоб швидко її побудувати з початкового масиву) і видаляти максимальний елемент (він поміщатиметься у вже відсортовану частину масиву - його правий кінець).

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, тобто форма піраміди має приблизно такий вигляд:



- виконується "властивість піраміди": кожний елемент менший, або дорівнює батькові.

Як зберігати таку послідовність? Найочевидніший і найпростіший спосіб - помістити її в масив.

Відповідність між геометричною структурою піраміди як дерева і масивом встановлюється за наступною схемою:

- в a[0] зберігається корінь дерева
- лівий і правий сини елемента a[i] зберігаються, відповідно, в a[2i+1] і a[2i+2]

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

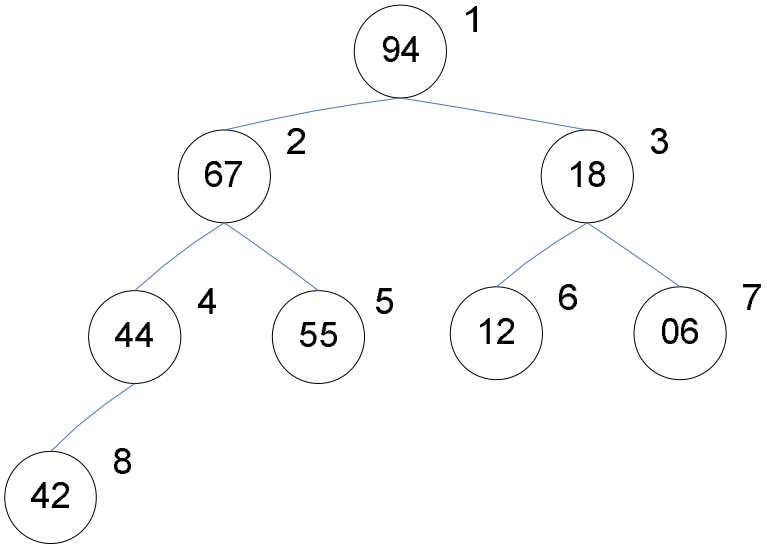


Рис.13 Приклад піраміди

Плюси такого зберігання піраміди очевидні:

- жодних додаткових змінних, необхідно лише розуміти схему;
- вузли зберігаються від вершини і далі вниз, рівень за рівнем;
- вузли одного рівня зберігаються в масиві зліва направо.

Запишемо у вигляді масиву піраміду, представлену вище. Зліва-направо, зверху-вниз: 94 67 18 44 55 12 06 42. На рисунку місце елемента піраміди в масиві позначено цифрою справа-вверху від нього.

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

1.6.1 Фаза 1 сортування: побудова піраміди

Розпочати побудову піраміди можна з a[k]...a[n], k = [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 межами масиву.

Слід зауважити, що невірно говорити про те, що a[k]..a[n] є пірамідою як самостійний масив. Це, взагалі кажучи, не вірно: його елементи можуть бути будь-якими. Властивість піраміди зберігається лише у рамках початкового, основного масиву a[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 елемента зберігалася пірамідальність, використовуватимемо наступну процедуру розширення піраміди a[i+1]..a[n] на елемент a[i] вліво:

1. Дивимося на синів зліва і справа - у масиві це a[2i+1] і a[2i+2] і вибираємо більшого з них.
2. Якщо цей елемент більше a[i] - міняємо його з a[i] місцями і йдемо до кроку 2,пам’ятаючи про нове положення a[i] у масиві. Інакше кінець процедури.

Новий елемент "просіюється" крізь піраміду.

Враховуючи, що висота піраміди h <= log n, алгоритм сортування вимагає O(log n) часу.

Нижче наведено ілюстрацію процесу сортування для піраміди з 8-ми елементів:

44 55 12 42 // 94 18 06 67 Справа - частина масиву, що

44 55 12 // 67 94 18 06 42 задовольняє властивості піраміди

44 55 // 18 67 94 12 06 42

44 // 94 18 67 55 12 06 42 решта елементів додається

// 94 67 18 44 55 12 06 42 один за іншим, справа наліво.

В геометричній інтерпретації ключі з початкового відрізка a[size/2]...a[n] є листками у бінарному дереві, як зображено нижче. Решта елементів один за другим просуваються на свої місця, і так доти, поки не буде побудована вся піраміда.

На рисунку нижче (рис.14) зображений процес побудови. Неготова частина піраміди (початок масиву) забарвлена у білий колір, що задовольняє властивості піраміди кінець масиву - в темний.

1.6.2. Фаза 2: власне сортування

Отже, задача побудови піраміди з масиву успішно вирішена. Як випливає із властивостей піраміди, у корені завжди знаходиться максимальний елемент. Звідси витікає алгоритм фази 2:

1. Беремо верхній елемент піраміди a[0]...a[n] (перший в масиві) і міняємо з останнім місцями. Тепер "забуваємо" про цей елемент і далі розглядаємо масив a[0]...a[n-1]. Для перетворення його в піраміду достатньо просіяти лише новий перший елемент.

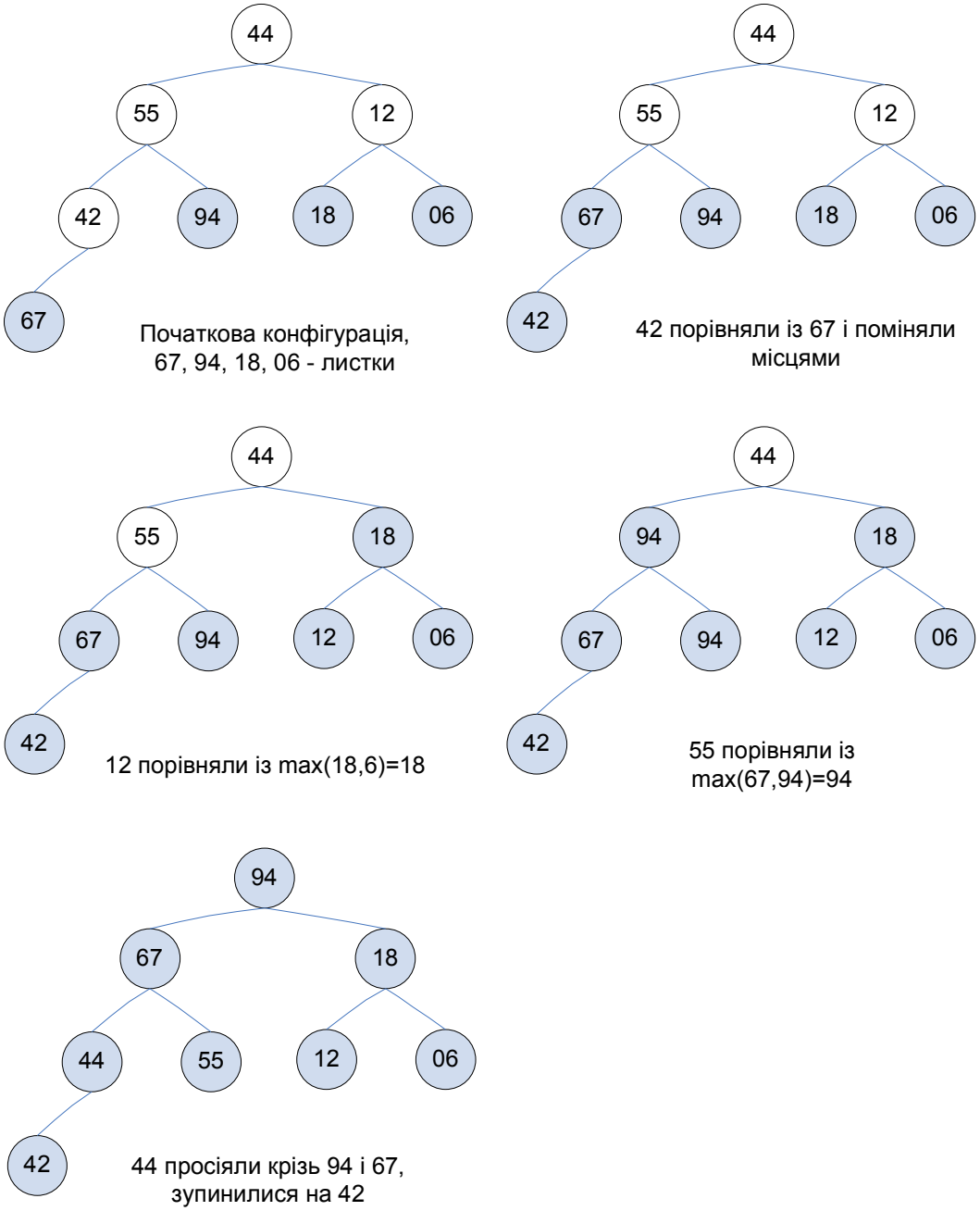


Рис.14 Етапи побудови піраміди з масиву

2. Повторюємо крок 1, поки оброблювана частина масиву не зменшиться до одного елемента.

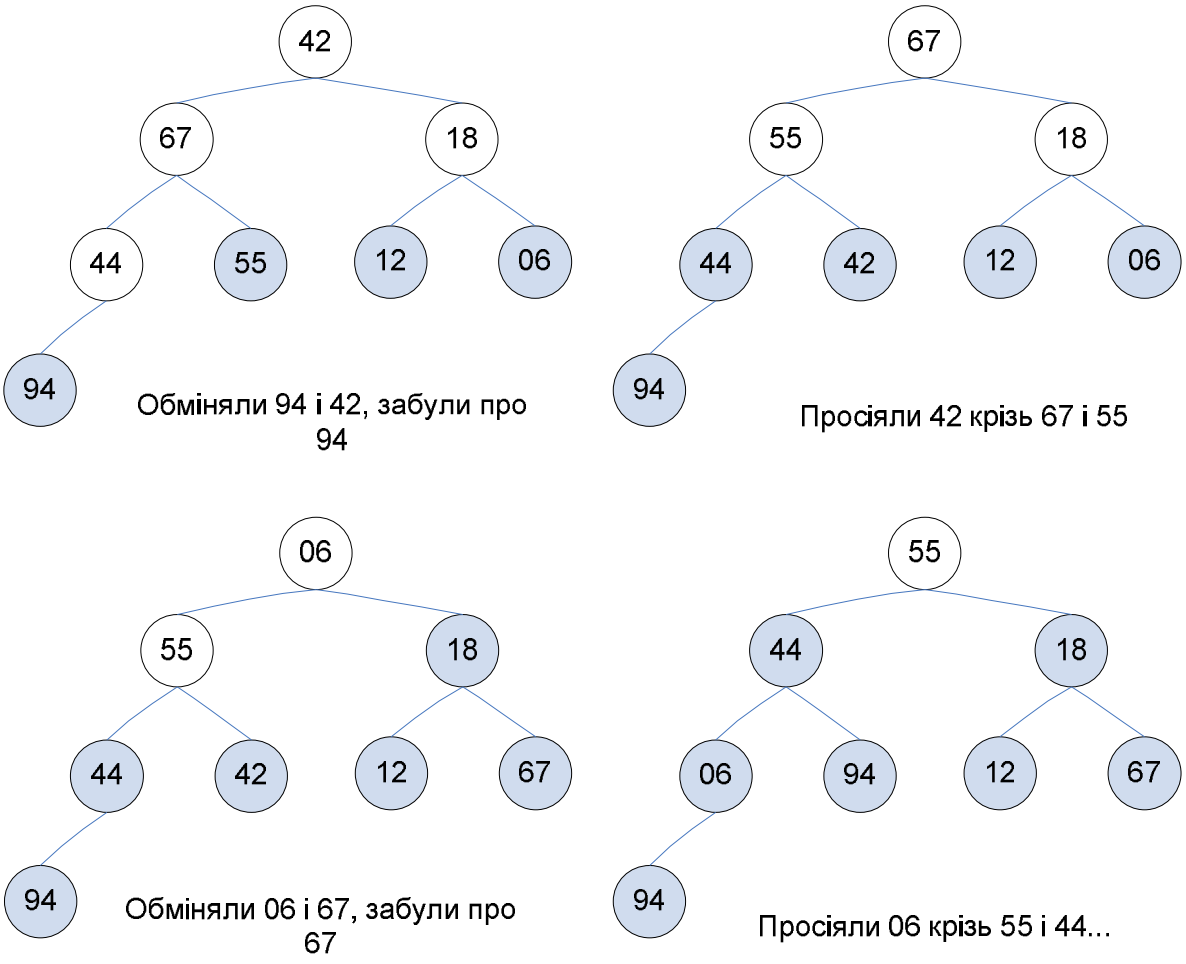


Рис.15 Приклад пірамідального сортування

Очевидно, що у кінець масиву кожного разу потрапляє максимальний елемент з поточної піраміди, тому у правій частині поступово виникає впорядкована послідовність. Ілюстрація 2-ї фази сортування у внутрішньому представленні піраміди:

94 67 18 44 55 12 06 42 //

67 55 44 06 42 18 12 // 94

55 42 44 06 12 18 // 67 94

44 42 18 06 12 // 55 67 94

42 12 18 06 // 44 55 67 94

18 12 06 // 42 44 55 67 94

12 06 // 18 42 44 55 67 94

06 // 12 18 42 44 55 67 94

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

Друга фаза займає O(n log n) часу: O(n) раз береться максимум і відбувається просіювання колишнього останнього елемента. Плюсом алгоритму є його стабільність: середня кідькість пересилок (n log n)/2, і відхилення від цього значення порівняно малі.

Пірамідальне сортування не використовує додаткової пам’яті.

Метод не є стійким: у процесі виконання роботи алгоритму масив так "перетрушується", що початковий порядок елементів може змінитися випадковим чином.

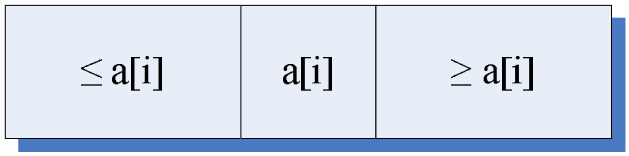
Поведінка алгоритму неприродна: часткова впорядкованість масиву жодним чином не враховується.

1.7 Швидке сортування

Алгоритм "швидкого сортування" був розроблений більше 40 років тому, і є найбільш широко вживаним і одним з найефективніших алгоритмів сортування. Метод заснований на підході "поділяй-і-владарюй". Загальна схема алгоритму наступна:

1. з масиву вибирається деякий опорний елемент a[i];
2. запускається процедура розділення масиву, яка переміщає всі ключі, які менше-рівні a[i], ліворуч від нього, а всі ключі, більші, або дорівнюють a[i] – вправо;

3. тепер масив складається з двох підмножин, причому ліва менша, або дорівнює правій;



4. для обох підмасивів: якщо у підмасиві більше двох елементів, рекурсивно запускаємо для нього ту саму процедуру.

У кінці роботи отримуємо повністю відсортовану послідовність. Розглянемо алгоритм детальніше.

1.7.1. Розділення масиву

На вході масив a[0]...a[N] і опорний елемент p, по якому проводитиметься розділення.

1. Введемо два покажчики: i і j. На початку алгоритму вони вказують, відповідно, на лівий і правий кінець послідовності.

2. Рухатимемо покажчик i з кроком в 1 елемент у напрямку до кінця масиву, поки не буде знайдений елемент a[i]>= p. Потім аналогічним чином почнемо рухати покажчик j від кінця масиву до початку, поки не буде знайдений a[j]<= p.

3. Далі, якщо i <= j, міняємо a[i] і a[j] місцями і продовжуємо рухати i,j за тими ж правилами.

4. Повторюємо крок 3, поки i <= j.

Розглянемо роботу процедури для масиву a[0]...a[6] і опорного елемента p = a[3].

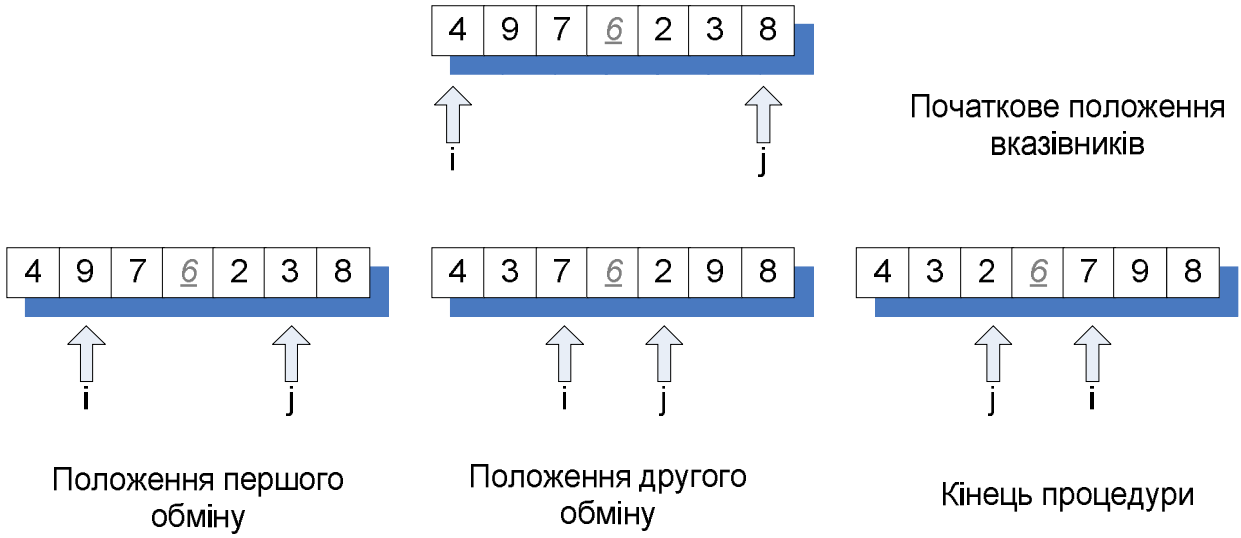


Рис.16 Приклад роботи алгоритму швидкого сортування

Тепер масив розділений на дві частини: всі елементи лівої менше або рівні p, всі елементи правої - більше, або рівні p. Розділення завершено.

1.7.2. Загальний алгоритм

Наведемо псевдокод алгоритму.

```
ШвидкеСортування ( масив a, верхня межа N ) {
   Вибрати опорний елемент p - середину масиву
   Розділити масив по цьому елементу
   Якщо підмасив зліва від p містить більше одного елемента
      викликати для нього ШвидкеСортування.
   Якщо підмасив праворуч від p містить більше одного елемента
      викликати для нього ШвидкеСортування.
}
```

Кожне розділення вимагає, очевидно, Theta(n) операцій. Кількість кроків розподілу (глибина рекурсії) складає приблизно log n, якщо масив ділиться на

більш-менш рівні частини. Таким чином, загальна швидкодія: O(n log n), що і має місце на практиці.

Проте, можливий випадок таких вхідних даних, на яких алгоритм працюватиме за $O(n^2)$ операцій. Таке відбувається, якщо кожного разу у ролі центрального елемент вибиратиметься максимум або мінімум вхідної послідовності. Якщо дані взяті випадково, ймовірність цього дорівнює 2/n. Ця ймовірність повинна реалізовуватися на кожному кроці.

Метод є нестійким. Поведінка досить природна, якщо врахувати, що при частковій впорядкованості підвищуються шанси розділення масиву на рівніші частини.

Сортування використовує додаткову пам’ять, оскільки приблизна глибина рекурсії складає O(log n), а дані про рекурсивні підвиклики кожного разу додаються у стек.

1.8 Порозрядне сортування

Алгоритм, який розглядатиметься у цьому розділі істотно відрізняється від описаних раніше. По-перше, він не використовує порівняння 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 gjxfnre сортування необхідно знати два параметри: k і m, де

- k - кількість розрядів у найдовшому ключі;
- m - розрядність даних: кількість можливих значень розряду ключа.

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

Аналогічно, для шістнадцяткових чисел m=16, якщо за розряд брати цифру, і m=256, якщо використовувати побайтовий розподіл.

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

1.8.1. Порозрядне сортування для списків

Припустимо, що елементи лінійного списку L є k -розрядні десяткові числа, і розрядність максимального числа відома наперед. Позначимо d(j,n) – j-ту справа цифру числа n, яку можна виразити як

d(j,n)= [n / 10j-1] % 10

Нехай L_0, L_1..., L_9 - допоміжні списки (кишені), спочатку вони порожні. Порозрядне сортування складається з двох процесів, які називаються відповідно розподіл і збирання і виконуються 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 дорівнюють m.

Фаза збирання полягає в об’єднанні списків L_0, L_1..., L_9 у загальний список $L = L_0 => L_1 => L_2 => ... => L_9$

Розглянемо приклад роботи алгоритму на вхідному списку 0 => 8 => 12 => 56 => 7 => 26 => 44 => 97 => 2 => 37 => 4 => 3 => 3 => 45 => 10.

Максимальне число містить дві цифри, значить, розрядність даних k=2.

Перший прохід, j=1.

Розподіл по першій справа цифрі:

L_0: 0 => 10 // всі числа з першою справа цифрою 0

L_1: порожньо

L_2: 12 => 2

L_3: 3 => 3

L_4: 44 => 4

L_5: 45

L_6: 56 => 26

L_7: 7 => 97 => 37

L_8: 8

L_9: порожньо // всі числа з першою справа цифрою 9

Збирання:

об’єднуємо списки Li один за іншим

L: 0 => 10 => 12 => 2 => 3 => 3 => 44 => 4 => 45 => 56 => 26 => 7 => 97 => 37 => 8

Другий прохід, j=2.

Розподіл по другій справа цифрі:

L0: 0 => 2 => 3 => 3 => 4 => 7 => 8

L1: 10 => 12

L2: 26

L3: 37

L4: 44 => 45

L5: 56

L6: порожньо

L7: порожньо

L8: порожньо

L9: 97

Збирання:

об’єднуємо списки Li один за другим

L: 0 => 2 => 3 => 3 => 4 => 7 => 8 => 10 => 12 => 26 => 37 => 44 => 45 => 56 => 97

Кількість використовуваних дорівнює кількості можливих значень елемента списку. Сортування можна організувати так, щоб не використовувати додаткової пам’яті для кишень, тобто не переміщати елементи списку, а за допомогою перестановки покажчиків приєднувати їх до тієї або іншої кишені.

1.8.2. Порозрядне сортування для масивів

Списки досить зручні тим, що їх легко реорганізовувати, об’єднувати тощо. Як застосувати аналогічну ідею для масивів?

Нехай у нас є масив source з n десяткових цифр (m = 10). Наприклад, source[7]= { 7, 9, 8, 5, 4, 7, 7 }, n=7. Нехай const k=1.

1. Створити масив count з m елементів(лічильників).

2. Присвоїти count[i] кількість елементів source, які дорівнюють i. Для цього:

   a. проініціалізувати count[] нулями,

   b. пройти по source від початку до кінця, для кожного числа збільшуючи елемент count з відповідним номером.

   c. for(i=0; i<n; i++) count [source[i]]++

   У нашому прикладі count[] = { 0, 0, 0, 0, 1, 1, 0, 3, 1, 1 }

3. Присвоїти count[i] значення, яке дорівнює сумі всіх елементів до даного:

count[i] = count[0]+count[1]+...count[i-1].

У нашому прикладі count[] = { 0, 0, 0, 0, 1, 2, 2, 2, 5, 6 } Ця сума є кількістю чисел початкового масиву, які менші за i.

4. Провести остаточну розстановку.

Для кожного числа source[i] ми знаємо скільки чисел менші за нього - це значення зберігається в count[source[i]]. Таким чином, нам відоме остаточне місце числа у впорядкованому масиві: якщо є K чисел менших за дане, тоді воно повинне стояти на позиції K+1.

Здійснюємо прохід по масиву source зліва направо, одночасно заповнюючи вихідний масив dest.

```
for ( i=0; i<n; i++ ) {
 c = source[i];
 dest[ count[c]]= з;
 count[c]++;  //  для чисел, що повторюються
}
```

Таким чином, число c=source[i] ставиться на місце count[c]. На випадок коли числа повторюються у масиві, передбачений оператор count[c]++, який збільшує значення позиції для наступного числа c, якщо таке буде.

Цикли займають (n + m) часу. Оцінимо потребу у пам’яті.

За час (n + m) алгоритм сортує цифри. А від цифр до рядків і чисел - 1 крок. Неай у нас у кожному ключі k цифр (m = 10). По аналогії до списків відсортуємо їх в декілька проходів від молодшого розряду до старшого.

Початкова k=3 послідовність	Перший прохід по 3-му розряду	Другий прохід по 2-му розряду	Третій прохід по 1-му розряду
523	523	523	088
153	153	235	153
088	554	153	235
554	235	554	523
235	088	088	554
	⇧	⇧	⇧
Offset (номер позиції)	3	2	1

Загальна кількість операцій, таким чином, (k(n+m)), при використовуваній додатково пам’яті (n+m). Ця схема допускає невелику оптимізацію. Зауважимо, що сортування по кожному байту складається з 2 проходів по всьому масиву: на першому кроці і на четвертому. Проте, можна створити відразу всі масиви count[] (по одному на кожну позицію) за один прохід. Неважливо, як розташовані числа -

лічильники не міняються, тому ця зміна коректна. Таким чином, перший крок виконуватиметься один раз за все сортування, а значить, загальна кількість проходів зміниться з 2k на k+1.

Ефективність порозрядного сортування.

Зростання швидкого сортування ми вже знаємо: O(n log n). Судячи з оцінки, порозрядне сортування зростає лінійно по n, оскільки k, m - константи. Також воно зростає лінійним чином по k - при збільшенні довжини типу (кількості розрядів) відповідно зростає кількість проходів.

Порозрядне сортування по своїй суті нецікаве для малих масивів. Кожний прохід містить мінімум 256 ітерацій, тому при невеликих n загальний час виконання (n+m) визначається константою m=256.

Поведінка неприродна, оскільки проходи виконуються повністю, незалежно від початкового стану масиву. Порозрядне сортування є стійким.

3. КОНТРОЛЬНІ ЗАПИТАННЯ

4. ЛАБОРАТОРНЕ ЗАВДАННЯ

5. ЗМІСТ ЗВІТУ

1. Мета роботи.
2. Теоретичний аналіз опрацьованого матеріалу.
3. Відповіді на контрольні запитання.
4. Індивідуальне завдання.
5. Аналіз отриманих результатів і висновки.
6. Список використаної літератури.

6. СПИСОК ЛІТЕРАТУРИ

7. ВАРІАНТИ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ

Програмно реалізувати

1. Сортування вибором
2. Бульбашкове сортування
3. ***Сортування вставками***
4. Сортування Шелла
5. Сортування злиттям
6. Пірамідальне сортування
7. Швидке сортування
8. Порозрядне сортування для списків
9. Порозрядне сортування для масивів

НАВЧАЛЬНЕ ВИДАННЯ

Алгоритми сортування.

Методичні вказівки
до лабораторної роботи № 10

з курсу “Теорія алгоритмів і структури даних”
для студентів базового напрямку 6.0804 - "Комп'ютерні науки"

Укладач *Керницький Андрій Богданович*

Редактор *Черничевич О.*