Масиви та рядки.
Алгоритми сортування масивів даних

Масив (**array**) – перелік змінних однакового типу, звернення до яких відбувається із застосуванням імені, загального для всіх його елементів.

Рядок (**string**) – клас з методами та змінними для організації роботи з рядками. Значеннями цього типу є довільна послідовність символів алфавіту. Кожна змінна такого типу може бути представлена фіксованою кількістю байтів або мати довільну довжину.

Поряд з числовими найчастіше використовуються символьні масиви, у яких зберігаються рядки. У мові програмування C++ не визначено вбудованого типу даних для зберігання рядків. Тому рядки реалізуються як масиви символів. Такий підхід до реалізації рядків дає C++- програмісту більше "важелів" керування порівняно з тими мовами, у яких використовується окремий рядковий тип даних.

Конкатенація рядків

Над рядками можна виконувати ряд операцій. Зокрема, можна об’єднувати рядки за допомогою стандартної операції складання:

```
string s1,s2, s3;
   s1 = "Hello";
   s2 = "world";
   s3 = s1 + " " + s2;
   cout << s3; // Hello world
```

Розмір рядка

За допомогою методу **size()** можна взнати розмір рядка, тобто із скількох символів він складається:

```
cout<<s1.size(); // 5
```

Отримання та зміна рядків

Подібно масиву ми можемо звертатися за допомогою індексів до окремих символів рядка, одержувати і змінювати їх:

```
cout<<s1[0]<<endl; // H
```

Одновимірні масиви

Одновимірний масив – перелік взаємопов'язаних між собою змінних.

Для оголошення одновимірного масиву використовують така форма запису:

тип ім'я_масиву[розмір];

Наприклад, у процесі виконання наведеної нижче настанови оголошується **int**-масив (що складається з 10 елементів) з іменем Array:

```
int Array[10];
```

У мові програмування C++ перший елемент масиву має нульовий індекс.

```
int Array[7];
for(int j=0; j<7; j++) Array[j] = j*j;
```

i[0]	i[1]	i[2]	i[3]	i[4]	i[5]	i[6]
0	1	4	9	16	25	36

```
list[3] = 10;
list[6] = 35;
list[5] = list[3] + list[6];
```

FIGURE Array `list` after execution of the statements `list[3]= 10;`, `list[6]= 35;`, and `list[5] = list[3] + list[6];`

Для одновимірних масивів загальний розмір масиву в байтах обчислюється так:
всього байтів = розмір типу елемента в байтах * кількість елементів.

У мові програмування С++ не можна присвоїти один масив іншому. У наведеному нижче коді програми, наприклад, присвоєння aMas = bMas; неприпустимо:

```
int aMas[10], bMas[10];
//...
aMas = bMas; // Помилка!!!
```

Щоб помістити вміст одного масиву в інший, необхідно окремо виконати присвоєння кожного значення:

```
int aMas[10], bMas[10], i;
//...
for(i=0; i<10; i++) aMas[i] = bMas[i]; // Правильно!
//...
```

У мові програмування С++ не здійснюється ніякої перевірки порушення контролю меж масивів, тобто нічого не може перешкодити програмісту звернутися до масиву за його межами. Якщо це відбувається у процесі виконання настанови присвоєння, то можуть бути змінені значення в елементах пам'яті, виділених деяким іншим змінним або навіть Вашій програмі.

Побудова символьних рядків

Найчастіше одновимірні масиви використовуються для побудови символьних рядків. У мові програмування С++ рядок визначається як символьний масив, який завершується нульовим символом ('\0'). Під час визначення довжини символьного масиву необхідно враховувати ознаку його завершення, тобто задавати його довжину на одиницю більше довжини найбільшого рядка, які передбачають зберігати у цьому масиві.

Рядок – символьний масив, який завершується нульовим символом.

Демонстрація механізму використання функції gets() для зчитування рядка з клавіатури

```
#include <iostream>     // Потокове введення-виведення
#include <cstdio>  // Підтримка системи введення-виведення
using namespace std;    // Використання стандартного простору імен
 int main()
   {
     char str[80];
     cout << "Введіть рядок: ";
     gets(str); // Зчитуємо рядок з клавіатури.
     cout << "Ось Ваш рядок: ";
     cout << str;
     return 0;
   }
```

Організація двовимірних масивів

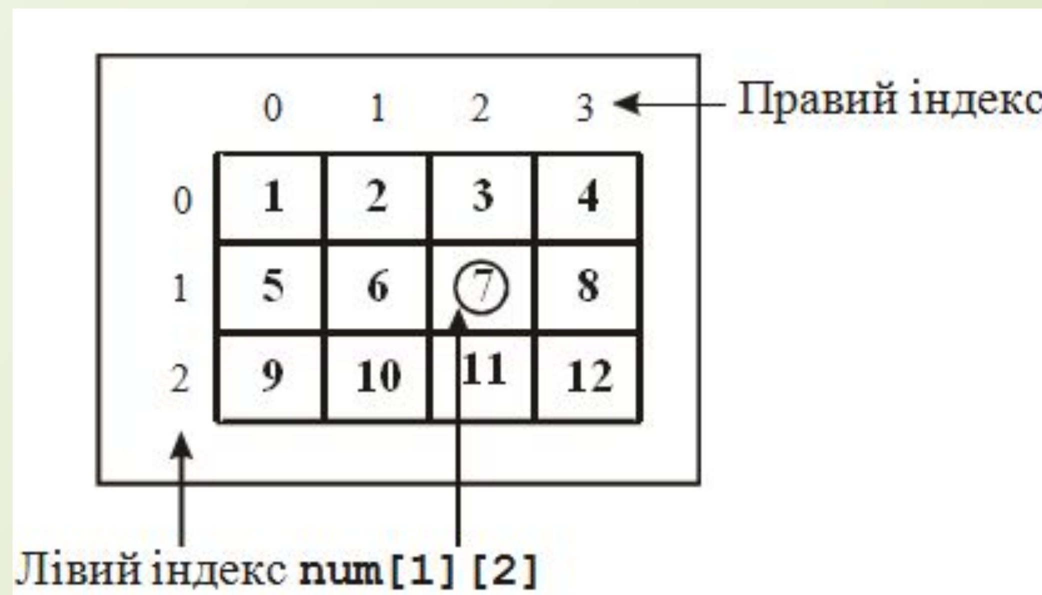
У мові програмування C++ можна використовувати двовимірні масиви. Двовимірний масив, по суті, є списком одновимірних масивів. Щоб оголосити двовимірний масив цілочисельних значень розміром 10х20 з іменем num, достатньо записати таке:

```
int num[10][20];
```

Щоб отримати доступ до елемента масиву num з координатами 3х5, необхідно використовувати запис num[3][5]. У наведеному нижче прикладі в двовимірний масив поміщаються послідовні числа від 1 до 12.

Демонстрація механізму роботи з елементами двовимірного масиву

```
#include <iostream>      // Потокове введення-виведення
using namespace std;     // Використання стандартного простору імен
 int main()
  {
    int t, i, num[3][4];
    for(t=0; t<3; ++t) {
     for(i=0; i<4; ++i) {
        num[t][i] = (t*4)+i+1;
        cout << num[t][i] << " ";
                              }
    cout << endl;
                          }
  return 0;
 }
```



Ініціалізація елементів масивів

Ознакою масиву при описі є наявність парних дужок []. Константа або константний вираз в квадратних дужках задає число елементів масиву. При описі масиву може бути виконана ініціалізація його елементів. Існує два методи ініціалізації елементів масивів: розмірних і безрозмірних масивів.

```
int Array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
char str[7] = "привіт";
int Array[10][2] = { 1, 1,
2, 4,
3, 9,
4, 16,
5, 25,
6, 36,
7, 49,
8, 64,
9, 81,
10, 100 };
```

Правий індекс

	0	1
0	1	1
1	2	4
2	3	9
3	4	16
4	5	25
5	6	36
6	7	49
7	8	64
8	9	81
9	10	100

Лівий індекс

Крім зручності в первинному визначенні масивів, метод "безрозмірної" ініціалізації дає змогу змінити будь-яке повідомлення без переліку його довжини. Тим самим усувається можливість внесення помилок, викликаних випадковим прорахунком.

```
char e1[] = "Ділення на 0\n";
char e2[] = "Кінець файлу\n";
char e3[] = "У доступі відмовлено \n";
```

У такому прикладі масив Array[][] оголошується як "безрозмірний":

```
int Array[][2] = { 1, 1,
2, 4,
3, 9,
4, 16,
5, 25,
6, 36,
7, 49,
8, 64,
9, 81,
10, 100
};
```

Число елементів масиву також можна визначати через константу:

```
const int n = 4;
int numbers [ n ] = { 1 , 2 , 3 , 4 };
```

Загальну кількість елементів масиву можна, наприклад, визначити так:

```
int size = sizeof (numbers ) / sizeof (numbers [0]);
```

Для знаходження довжини застосовується оператор **sizeof**. По суті довжина масиву рівна сукупній довжині його елементів. Всі елементи представляють один і той же тип і займають один і той же розмір в пам'яті. Тому за допомогою виразу **sizeof(numbers)** знаходимо довжину всього масиву в байтах, а за допомогою виразу **sizeof(numbers[0])** - довжину одного елементу в байтах. Розділивши два значення, можна одержати кількість елементів в масиві.

Є і ще одна форма циклу **for**, яка призначена спеціально для роботи з колекціями, у тому числі і з масивами. Ця форма має наступне формальне визначення:

for (змінна : колекція) { і н с т р у к ції ; }

Використовуємо цю форму для перебору масиву:

```
#include <iostream>;
int main ( )
{
  int numbers [4] = {1 , 2 , 3 , 4};
  int number;
    for (number : numbers)
       cout << number << endl;
return 0;
}
```

При переборі масиву кожний перебраний елемент поміщатиметься в змінну **number**, значення якої в циклі виводиться на консоль.

Також для перебору елементів багатовимірного масиву можна використовувати іншу форму циклу for:

```
int number, subnumbers;
    const int rows = 3, columns = 2;
int numbers [rows] [columns] = { {1, 2}, {3, 4}, {5, 6} };
   for ( subnumbers : numbers)
    {
        for (number : subnumbers)
            {
                cout << number << "\t" ;
            }
        cout << endl;
    }
```

Алгоритми сортування масивів даних

Сортування є однією з фундаментальних алгоритмічних задач програмування.

Сортування - це процес перегрупування заданої множини об’єктів, що приводить до їх впорядкованого розташування щодо ключа. Мета сортування - полегшити подальший пошук елементів.

В алгоритмах сортування лише частина даних використовується в якості ключа. **Ключ сортування** – атрибут (або декілька атрибутів), за значенням якого визначається порядок сортування.

Однією важливою властивістю алгоритмів сортування є їх сфера застосування. За цим параметром алгоритми розділяють на дві основні групи: - ***внутрішнє сортування*** (працюють з даними в оперативній пам’яті з довільним доступом); ***Зовнішнє сортування*** (впорядковують інформацію, розташовану на зовнішніх носіях).

Універсального, найкращого алгоритму сортування на даний момент не існує. Однак, маючи приблизні характеристики вхідних даних, можна підібрати метод, що працює оптимальним чином.

Оцінка алгоритмів сортування проводиться за наступними параметрами:

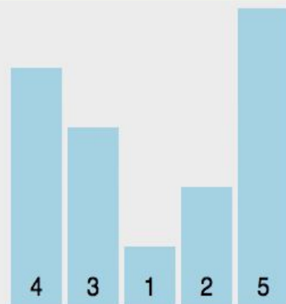
- **час сортування** (параметр, що характеризує швидкодію алгоритму);
- **пам’ять** (параметр, який характеризується тим, що ряд алгоритмів сортування вимагають виділення додаткової пам’яті під тимчасове зберігання даних);
- **стійкість** (параметр, який відповідає за те, що сортування не змінює взаємного розташування рівних елементів);
- **природність поведінки** (параметр, який вказує на ефективність методу при обробці вже відсортованих або частково відсортованих даних; алгоритм поводиться природно, якщо враховує цю характеристику вхідної послідовності, і працює краще).

Найбільш поширені алгоритми сортування: бульбашкове (bubble sort), коктейльне (cocktail sort), гребінцем(comb sort), швидке(quick sort), простими вставками (insert sort), сортування Шелла (Shell sort), злиттям (merge sort), вибором (selection sort), пірамідальне (heap sort), тощо.

Бульбашкове сортування

Інші назви: сортування простим обміном.
Клас сортування: обміном.
Стійкість: так.
Порівняння: так.
Складність по часу:
- найгірша: $0(n^2)$;
- середня: $0(n^2)$;
- - краща: 0(n).

Складність по пам'яті:
- загальна: O(n);
- додаткова: 0(1).



Сортування бульбашкою - найпростіший алгоритм сортування, застосовуваний тільки для навчальних цілей. Практичного застосування цього алгоритму немає, так як він не ефективний, особливо якщо необхідно відсортувати масив великого розміру. До плюсів сортування бульбашкою відноситься простота реалізації алгоритму.

Алгоритм складається з повторення проходів масивом, що сортується. За кожен прохід елементи послідовно порівнюються попарно і, якщо порядок в парі неправильний, виконується обмін елементів. Проходи масивом повторюються до тих пір, поки на черговому проході не виявиться, що обміни більше не потрібні. Це означає, що масив відсортований. Під час роботи алгоритму елемент, що розташований не на своєму місці, ніби «спливає» до потрібної позиції, як бульбашка у воді, звідси і назва алгоритму.

Бульбашкове сортування: 1 крок

Алгоритм сортування методом бульбашки: порівнюємо поточний і наступний елементи масиву. Якщо поточний елемент більший, ніж наступний, то міняємо їх місцями.

В результаті останнім елементом в масиві у нас виявиться найбільший елемент.

Бульбашкове сортування: 2 крок

Другий крок сортування методом бульбашки — повторюємо вищевказані дії для частини масиву, починаючи з **1** позиції до ***N*-1**. В результаті передостанній елемент в масиві теж буде на своєму, "відсортованому" місці.

На наступних кроках сортування методом бульбашки вищевказані дії повторюються для частини масиву, починаючи з **1** позиції до ***N*-2** (крок 3), а потім для діапазону **1**...***N*-3** і так далі до діапазону **1**...**2**.

Після завершення останнього кроку масив буде відсортований за зростанням.

	6	9	4	1	5	3	7
i=0	6	4	1	5	3	7	9
i=1	4	1	5	3	6	7	9
i=2	4	1	3	5	6	7	9
i=3	1	3	4	5	6	7	9

Звідси можна зробити висновок, що алгоритм бульбашки досить повільний, проте він простий і його можна поліпшити простими засобами.

По-перше, розглянемо ситуацію, коли на якому-небудь з проходів не відбулося жодного обміну. Це означає, що всі пари розташовані в правильному порядку, так що масив вже відсортований. Одне з можливих поліпшень алгоритму полягає в запам’ятовуванні, чи проводився на даному проході який-небудь обмін. Якщо ні — алгоритм закінчує роботу.

Процес поліпшення можна продовжити, якщо запам’ятовувати не тільки сам факт обміну, а й індекс останнього обміну ***k***. Дійсно: всі пари сусідніх елементів з індексами, більшими ***k***, вже розташовані в потрібному порядку. Подальші проходи можна закінчувати на індексі ***k***, замість того щоб рухатися до встановленої заздалегідь верхньої межі ***i***.

Коктейльне сортування (cocktail sort)

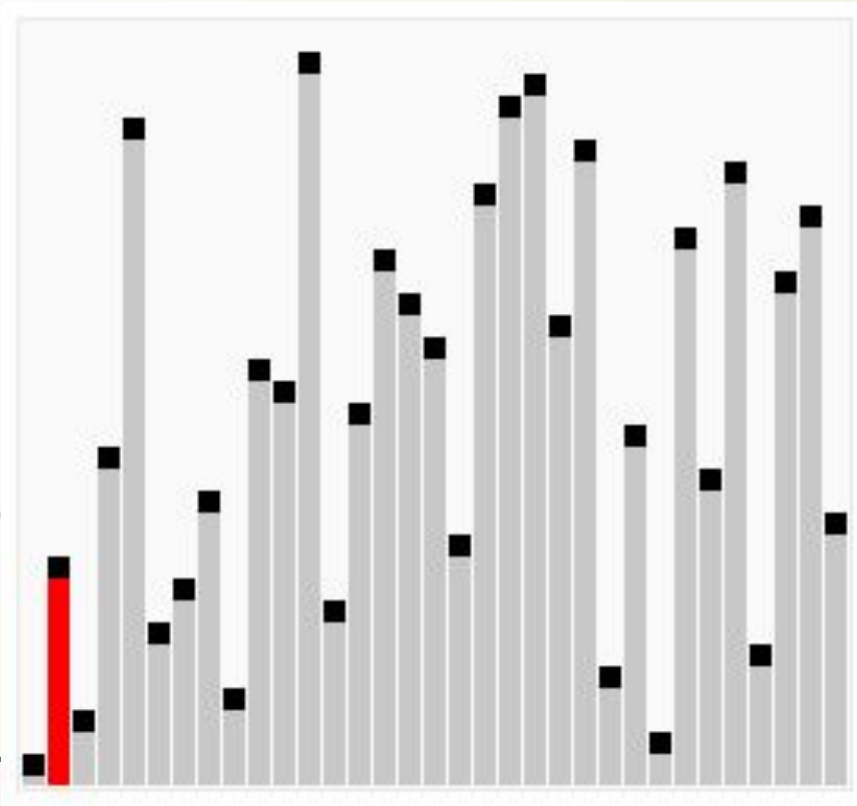
Інші назви: шейкерне сортування (shaker sort), сортування перемішуванням (shuffle sort), човникове сортування (shuttle sort), пульсуюче сортування (ripple sort), двостороннє бульбашкове сортування (bidirectional bubble sort).

Проводиться багаторазовий прохід масивом, при цьому сусідні елементи порівнюються і, в разі необхідності, міняються місцями.

Клас сортування: обміном.
Стійкість: так.
Порівняння: так.
Складність по часу:
- найгірша $0(n^2)$;
-середня $0(n^2)$;
-краща 0(n).
Складність по пам'яті:
- загальна 0(n);
-додаткова 0(1)

Сортування змішуванням мало чим відрізняється від сортування бульбашкою. Єдина його відмінність у тому, що замість багаторазового проходження через список знизу вгору, він проходить по черзі знизу вгору і згори вниз. Він може досягати трохи вищої ефективності, ніж алгоритм сортування бульбашкою. Причиною цьому є те, що алгоритм сортування бульбашкою проходить по списку лише в одному напрямі, а тому за одну ітерацію елементи списку можна перемістити лише на один крок.

Наприклад, для того, щоб відсортувати список (2,3,4,5,1), алгоритму сортування змішуванням достатньо лише одного проходу, у той час, як алгоритму сортвування бульбашкою знадобиться чотири проходи.



Демонстрація у форматі gif.

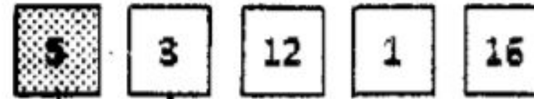
Кращий випадок для коктейльного сортування – це відсортований масив (O(n)), найгірший – відсортований у зворотному порядку ($O(n^2)$).

Найменше число порівнянь в алгоритмі **N-1**. Це відповідає єдиному проходу по впорядкованому масиву (кращий випадок).

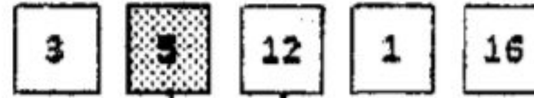
Алгоритм зважає на те, що від останньої перестановки до кінця (початку) масиву знаходяться відсортовані елементи. З огляду на цей факт, перегляд здійснюється не до кінця (початку) масиву, а до конкретної позиції. Тому під час коктейльного сортування необхідно запам'ятовувати індекс останньої перестановки. На наступному кроці перегляд масиву починається з позиції останньої перестановки. Перегляд масиву здійснюється зліва направо (встановлюється права межа). Потім справа наліво (встановлюється ліва межа). Перегляд масиву здійснюється до тих пір, поки всі елементи не встануть в порядку збільшення (зменшення).

Початковий масив:

Крок 1:

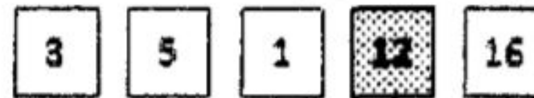


5>3 → true, обмін

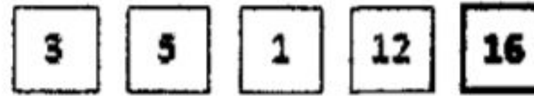


5>12 → false

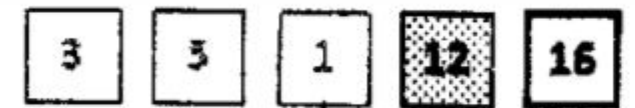
12>4 → true, обмін



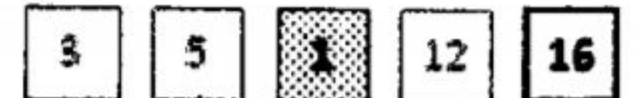
12>16 → false



Крок 2:

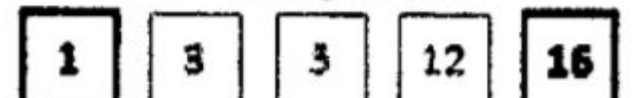


12<1 → false



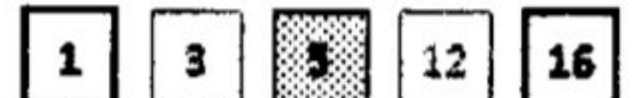
1<5 → true, обмін

1<3 → true, обмін

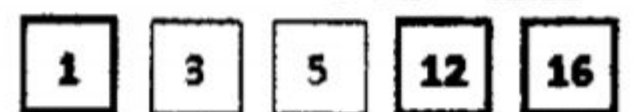


Крок 3:

3>5 → false



5>12 → false



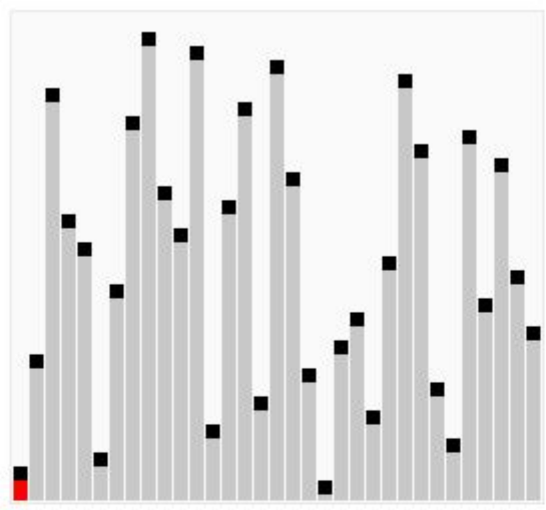
Крок 4:

3>5 → false

Результат сортування:

Сортування гребінцем (comb sort)

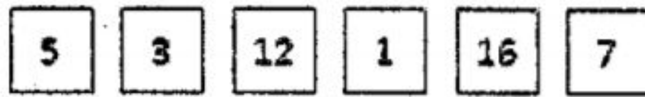
Клас сортування: обміном. *Стійкість*: так. *Порівняння*: так.



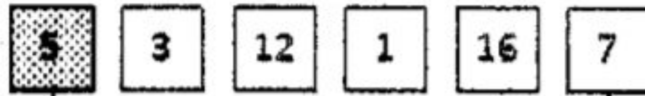
Алгоритм представляє собою модифікацію бульбашкового сортування, в якому порівняння і перестановки елементів масиву відбуваються на деякій фіксованій відстані один від одного - на проміжку **gap**. Для першого проходу масивом проміжок **gap** є максимальним, а для кожного наступного проходу значення **gap** зменшується або арифметично - на деяку величину, що називається **кроком**, або геометрично - в деяке число разів, що називається **усадковим фактором**. Після досягнення значення **gap = 1** сортування гребінцем вироджується в звичайне бульбашкове сортування. Бульбашкове сортування буде тривати, доки масив не буде повністю опрацьований. Тобто прохід бульбашкового сортування, при якому не буде зафіксовано жодного обміну значень між елементами масиву, є останнім.

Вважається, що для сортування гребінцем з геометричним зменшенням **gap** найкращим є усадковий фактор, що дорівнює **3/4**. Однак операції ділення та множення затратні для процесора, навіть за умови їх виконання в зовнішньому циклі. Звідси очевидно, що частина вигоди від геометричного зменшення проміжку **gap** під час проведення сортування нівелюється за рахунок особливостей обчислювальних машин.

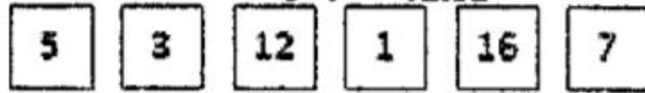
Початковий масив:



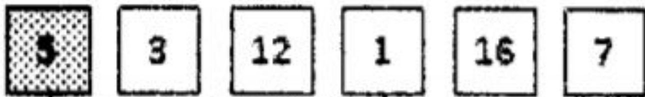
Крок 1 (*gap* = *5*):



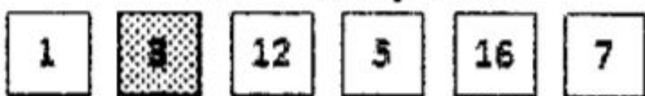
5>7 → false



Крок 2 (*gap* = *3*):



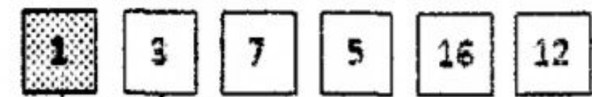
5>1 → true, обмін



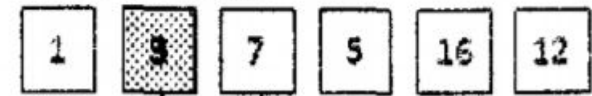
3>16 → false

12>7 → true, обмін

Крок 3 (*gap* = *1*):



1>3 → false

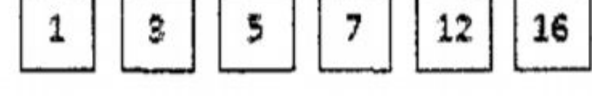


3>7 → false

7>5 → true, обмін

7>16 → false

16>12 → true, обмін



Результат сортування:

Головним недоліком сортування гребінцем з геометричним зменшенням gap є імовірність потрапляння в область невдалих обмінів, що відбувається під час певного збігу значень довжини масиву і розташування чисел в ньому. Тоді масив меншого розміру може сортуватися цим методом в десятки разів повільніше порівняно з сортуванням масиву більшого розміру.

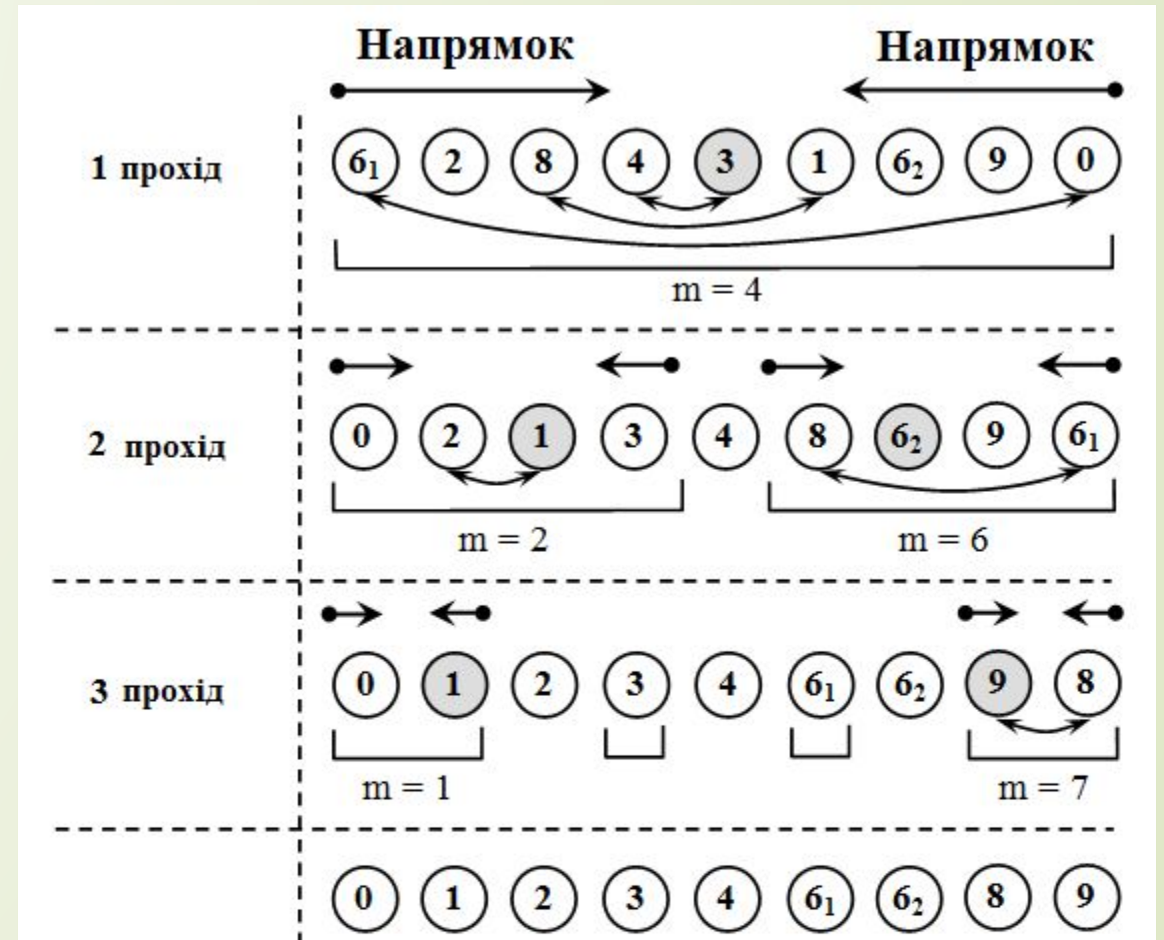
Швидке сортування (quick sort) (Сортування Хоара)

Швидке сортування використовує стратегію «розділяй і володарюй».

Клас сортування: обміном.
Стійкість: ні.
Порівняння: так.
Складність по часу:
- краща 0(n);
- середня 0(n*log n);
- найгірша 0 (n^2)

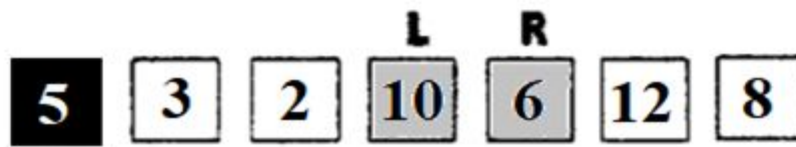
Спочатку з елементів масиву обрати деяке значення як ***опорний елемент*** і переставити елементи масиву так, щоб елементи зліва від опорного були не більше нього, а елементи справа від опорного — не менше. Далі процедура швидкого сортування застосовується до кожної з одержаних частин масиву.

Даний алгоритм, хоча і є найшвидшим з відомих, однак не є легким для аналізу і розуміння, на відміну від простіших алгоритмів на подобі алгоритму простих вставок або алгоритму вибору – його буде важче реалізувати по пам’яті ніж інші.



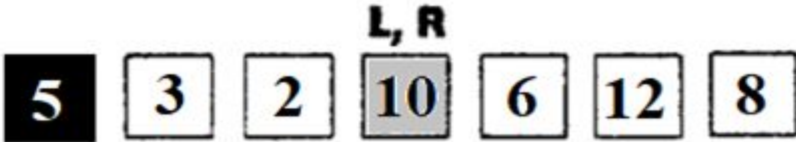
Початковий масив:	5	3	8	10	6	12	2	
Визначення опорного елементу (наприклад, 1-й елемент масиву):	**5**	3	8	10	6	12	2	
Крок 1:	**5**	3 (L)	8	10	6	12	2 (R)	L: 3>5 → false R: 2<5 → true R>L → false
Крок 2:	**5**	3	8 (L)	10	6	12	2 (R)	L: 8>5 → true R: 2<5 → true R>L → false, обмін R з L
Крок 3:	**5**	3	2	10 (L)	6	12 (R)	8	L: 10>5 → true R: 12<5 → false R>L → false

Крок 4:



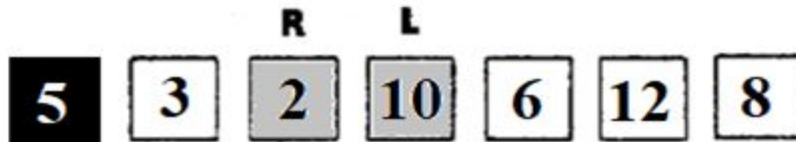
L: 10>5 → true
R: 6<5 → false
R>L → false

Крок 5:



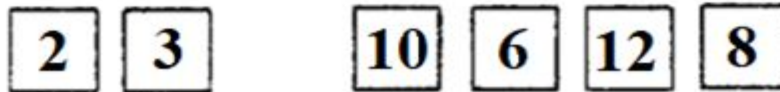
L: 10>5 → true
R: 10<5 → false
R>L → false

Крок 6:



L: 10>5 → true
R: 2<5 → true
R>L → true, обмін з R

Положення R розбиває масив на два підмасиви, до яких рекурсивно застосовують ті ж самі кроки алгоритму швидкого сортування:



... ...

Результат сортування:

Швидке сортування є самим швидкодіючим з усіх існуючих алгоритмів сортування обміном. Швидше нього тільки спеціалізовані алгоритми, що використовують специфіку даних, які сортуються. До переваг алгоритму також відноситься простота реалізації і хороше поєднання з алгоритмами кешування і підкачки пам’яті.

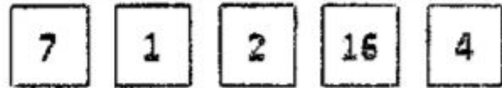
Недоліком алгоритму є його рекурсивність, яка викликає необхідність витрачати пам’ять комп’ютера на запис адрес повернення з підпрограми сортування кожного з підмасивів.

Сортування простими вставками (insert sort)

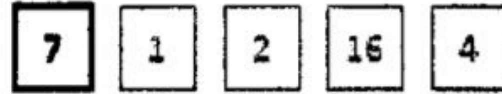
Клас сортування: вставками. *Стійкість*: так. *Порівняння*: так.

У сортуванні вставками масив, що сортується, можна розділити на дві частини – відсортована частина і несортована. На початку сортування перший елемент масиву вважається відсортованим, все інші – невідсортовані. Починаючи з другого елемента масиву і закінчуючи останнім, алгоритм вставляє невідсортований елемент масиву в потрібну позицію у вже відсортованій частині масиву. Для операції вставки використовується буферна область пам'яті, в якій на даний момент зберігається елемент, ще не вставлений на своє місце у відсортованому масиві. Цей елемент називається **ключовим елементом**. У відсортованій частині масиву, починаючи з її кінця, один за одним перебираються елементи і порівнюються з ключовим. Якщо ці елементи більше ключового, то вони зсуваються на одну позицію вправо, звільняючи місце для наступних елементів. Якщо в результаті перебору зустрічається елемент, менший або рівний ключовому, то в цьому випадку в поточну вільну комірку вставляється ключовий елемент. Таким чином, на кожній ітерації розглядається тільки один елемент невідсортованої частини масиву і шукається його місце у вже відсортованій частині.

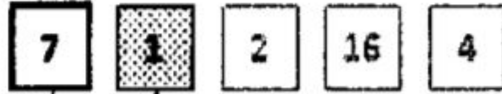
Початковий масив:



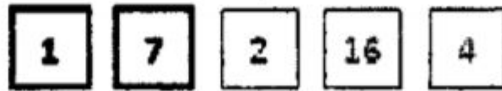
Крок 1:



Крок 2:



7>1 → true, обмін



Крок 3:

7>2 → true, обмін

1>2 → false

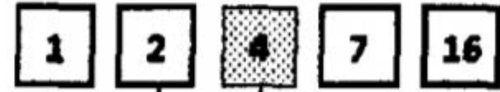
Крок 4:

7>16 → false

Крок 5:

16>4 → true, обмін

7>4 → true, обмін



2>4 → false

Результат сортування:

До позитивної сторони методу відноситься простота реалізації, а також його ефективність на частково впорядкованій послідовності. Основною перевагою алгоритму сортування вставками є можливість сортувати масив у міру його отримання. Тобто, маючи частину масиву, можна починати його сортувати. Проте висока обчислювальна складність не дозволяє рекомендувати алгоритм в повсюдному використанні.

Сортування Шелла

Сортування Шелла приблизно так само отримується із сортування вставками, як сортування гребінцем із бульбашкового.

Клас сортування: вставками. *Стійкість*: ні.
Порівняння: так.
Складність по часу:
- найгірша: залежить від кроку;
- середня: залежить від кроку;
- краща: O(n).

Складність по пам'яті:
- загальна: O(n);
- додаткова: 0 (1).

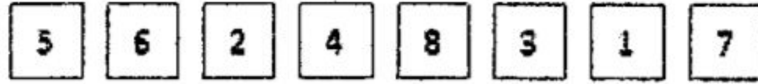
Ідея алгоритму полягає в порівнянні розділених на групи елементів масиву, що знаходяться один від одного на деякій відстані **d**. Спочатку ця відстань дорівнює **N/2**, де **N** - загальне число елементів масиву. Таким чином, на першому кроці в алгоритмі будуть попарно порівнюватися між собою і, в разі необхідності, мінятися місцями елементи, що розташовані один від одного на відстані **N/2**, тобто впорядковуються **N/2** пари елементів. На наступних кроках також відбуваються перевірка і обмін, але відстань **d** скорочується на **d/2**, а відтак зменшується і кількість груп елементів, які порівнюються. Тобто на другому кроці будуть впорядковуватися елементи вже в **N/4** групах, кожна з яких буде містити по чотири елементи для впорядкування. Поступово відстань між елементами зменшується, і на кроці, коли **d=1**, прохід по масиву відбувається в останній раз, а впорядкування елементів відбувається одразу в усьому масиві. На кожному кроці для впорядкування елементів в межах групи використовується сортування вставками.

Величина кроку **d** називається **приростом** і є важливою характеристикою алгоритму Шелла. Вибір динаміки зменшення цієї величини дуже істотно позначається на продуктивності алгоритму в цілому.

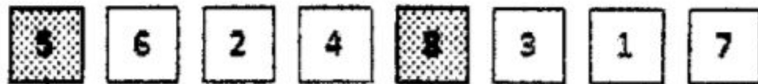
Розбити масив на групи елементів, що знаходяться на певній відстані один від одного, і здійснити незалежне сортування цих груп (як правило, методом вставки). На кожній ітерації крок між елементами групи зменшується і на останній ітерації він дорівнює одиниці. Складність сортування залежить від способу вибору кроку.

Сортування вставками є ефективним для обробки майже відсортованих масивів. Сортування Шелла при початкових проходах є доволі швидким, воно призводить масив до стану певної часткової впорядкованості. На заключному етапі крок дорівнює одиниці, тобто сортування Шелла природним чином трансформується в сортування простими вставками.

Початковий масив:

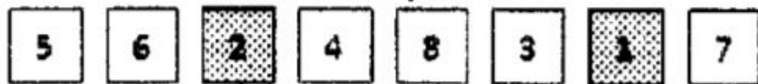


Крок 1 (*d* = *4*):



5>8 → false

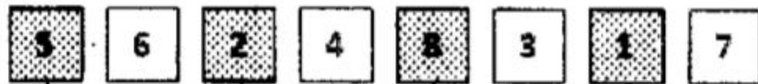
6>3 → true, обмін

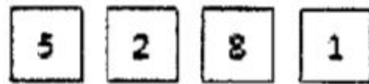


2>1 → true, обмін

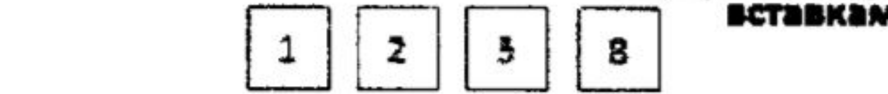
4>7 → false

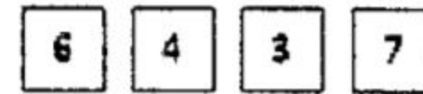
Крок 2 (*d* = *2*):



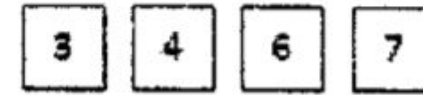


сортування простими вставками

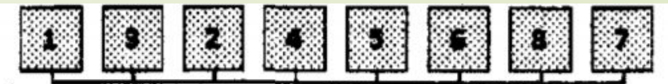


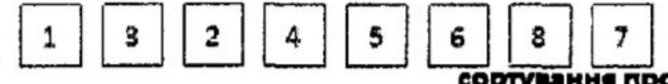


сортування простими вставками

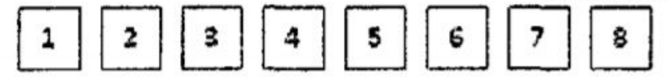


Крок 3 (*d* = *1*):





сортування простими вставками

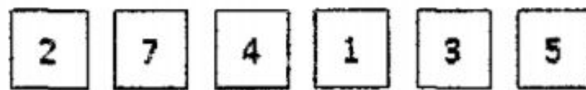


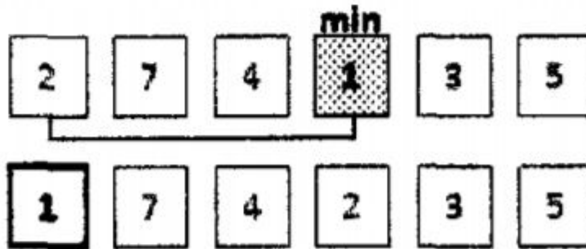
Результат сортування:

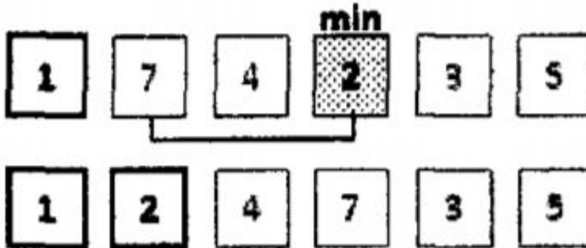
Сортування вибором (selection sort)

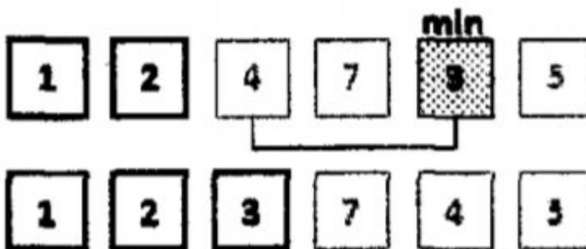
Клас сортування: вибором. *Стійкість*: ні. *Порівняння*: так.

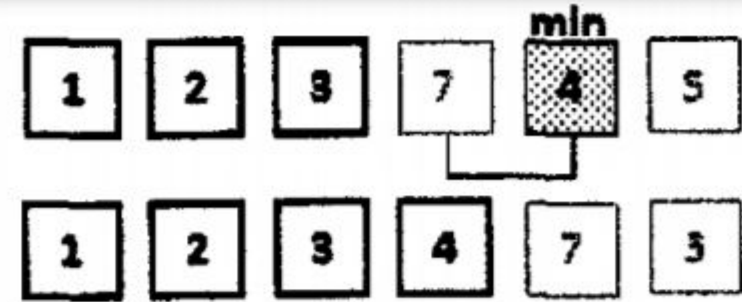
Ідея методу полягає в тому, щоб створювати відсортовану послідовність шляхом приєднання до неї елементів в правильному порядку один за одним. На початку алгоритму в масиві знаходять максимальний або мінімальний елемент, в залежності від того, як необхідно відсортувати масив (за збільшенням або за зменшенням). У випадку, якщо необхідно відсортувати масив за збільшенням його елементів, знаходять максимальний елемент. Знайдений максимальний елемент міняють місцями з останнім елементом. Потім виконують прохід за невідсортованою частиною масиву (від першого елемента до передостаннього) і вже в цій частині знаходять максимум, який потім міняють місцями з передостаннім елементом масиву. Таким чином продовжуються пошук і обміни, поки масив не буде повністю відсортовано.

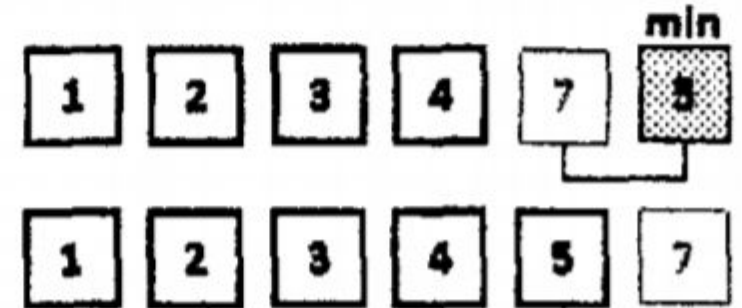












Недолік сортування вибором полягає в тому, що час його виконання лише в незначній мірі залежить від того, наскільки впорядкований масив перед початком сортування. Процес знаходження шуканого елемента за один прохід файлу дає дуже мало відомостей про те, де може перебувати елемент на наступному проході цього самого файлу.

Пірамідальне сортування

Клас сортування: вибором.
Стійкість: ні.
Порівняння: так.
Складність по часу:
-краща: 0(n*log n). *Складність по пам'яті*:
- загальна: 0(n);
- додаткова: 0(1).

6 5 3 1 8 7 2 4

Пірамідальне сортування складається з двох етапів: формування з початкового масиву такої структури даних, як двійкова купа, і виконання безпосереднього сортування.

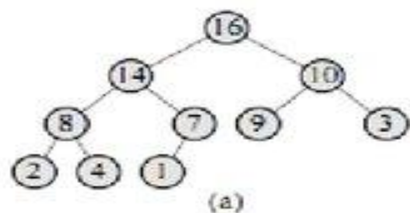
Двійкова купа - це двійкове дерево, для якого виконуються умови:
- значення в будь-якій вершині не менше, ніж значення її нащадків;
- глибина листя (відстань до кореня) відрізняється не більше, ніж на один рівень;
- останній рівень заповнюється зліва направо.

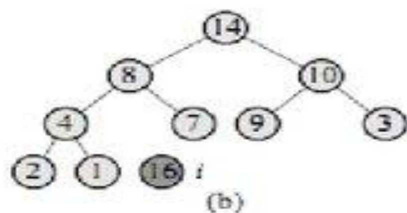
Для побудови двійкової купи початковий масив ділиться навпіл, при цьому друга його половина вже приймається за правильно побудовану двійкову купу. Потім послідовно беруться елементи з першої половини і додаються в двійкову купу на потрібні місця. Дійсно, для другої половини початкового масиву основна властивість двійкової купи виконується автоматично. Вірніше буде сказати, що ця властивість не порушується, оскільки для елементів другої половини просто не існує нащадків.

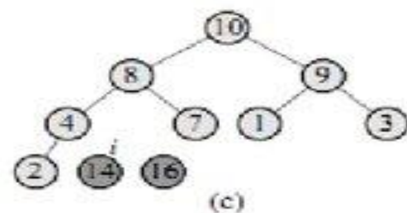
На етапі додавання елементів в першу половину двійкової купи, щоб виконати додавання елемента, потрібно поміняти його з найбільшим із нащадків, якщо останній перевершує значення елемента. Потім те ж саме необхідно виконати по відношенню вже до нових його нащадків.

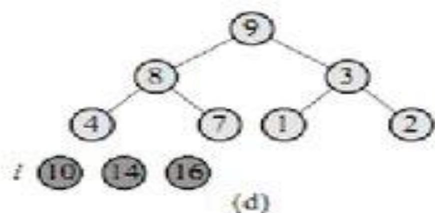
Після побудови двійкової купи запускається процедура безпосереднього сортування. Для цього з двійкової купи вилучається корінь і ставиться на початку майбутньої відсортованої послідовності. На місце вилученого елемента поміщаєіься кінцевий елемент двійкової купи, після чого необхідно відновити правильну купу. Ідея такого відновлення полягає в тому, що, якщо основна властивість у новому кореневому елементі не виконується, то більший з нащадків обмінюється з предком, після чого основна властивість перевіряється і у нащадка. Як результат, в корені двійкової купи знову опиниться максимальний елемент. На другому кроці новий корінь знову переміщається в кінцеву послідовність і т.д. до тих пір, доки вся двійкова купа не скінчиться.

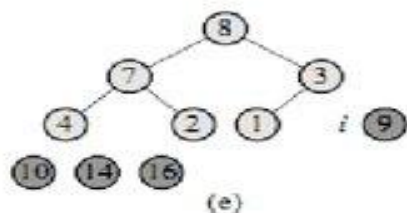
Алгоритм пірамідального сортування

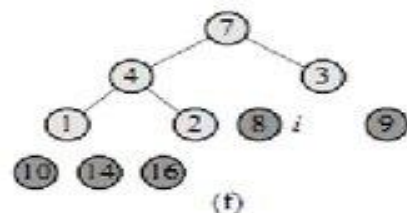


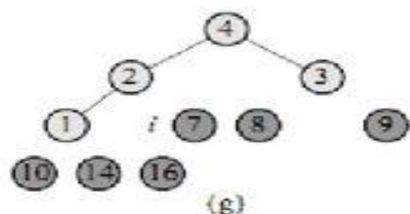


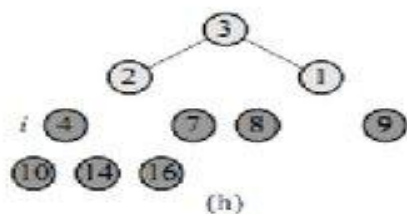


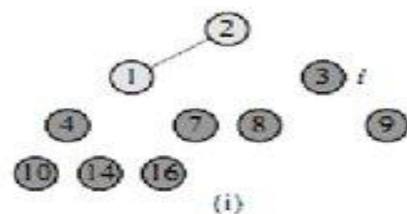


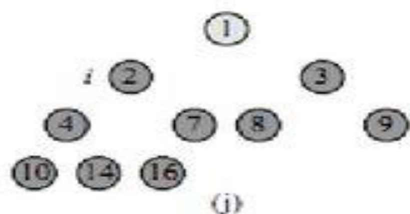












A	1	2	3	4	7	8	9	10	14	16

(k)