Алгоритми сортування.
Частина 2

Навчальні питання:

1. Сортування вставками (*Insertion Sort*):
    1.1. Сортування простими вставками.
    1.2. Па́рне сортування простими вставками.
    1.2. Сортування простими вставками з бінарним пошуком.
2. Сортування Шелла.

1. Сортування вставками (*Insertion Sort*)

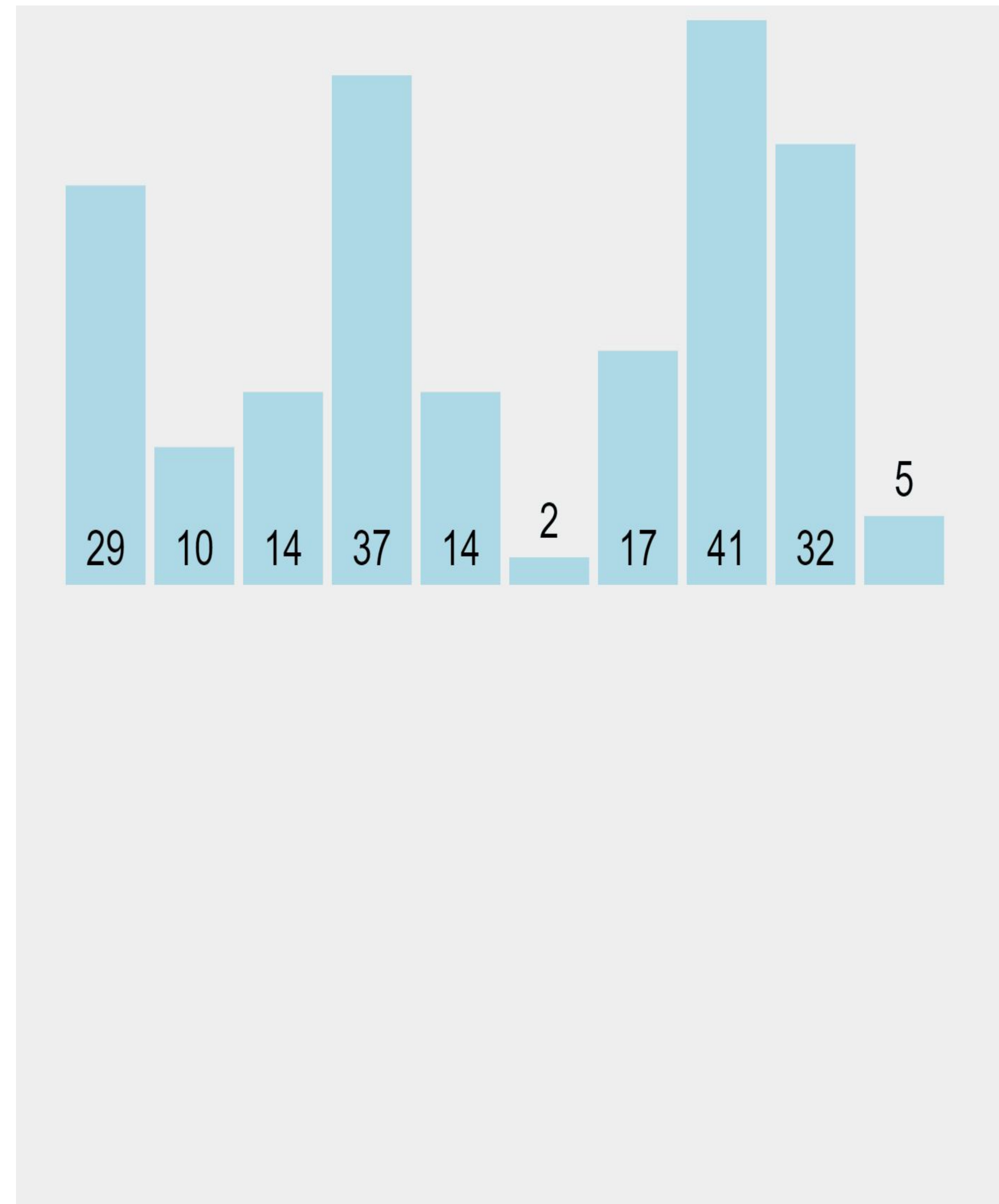
Сортування вставками (*Insertion Sort*) – це простий алгоритм сортування. Суть його полягає в тому, що на кожному кроці алгоритму ми беремо один з елементів масиву, знаходимо позицію для вставки та вставляємо. Масив з одного елемента вважається відсортованим.

Словесний опис алгоритму звучить досить складно, але насправді це найпростіше у реалізації сортування.
Кожен із нас, незалежно від роду діяльності, застосовував алгоритм сортування, просто не усвідомлював це:) Наприклад, коли сортували купюри в гаманці — беремо 100 грн і дивимося — йдуть 10, 50 і 500 гривневі купюри. Ось якраз між 50 і 500 і вставляємо нашу сотню:)
Або наведу приклад із усіх книжок – гра в карткового «Дурня». Коли ми тягнемо карту з колоди, дивимося на наші розкладені за зростанням карти та залежно від гідності витягнутої карти поміщаємо картку у відповідне місце.

У нашому прикладі, починаючи зліва, алгоритм позначає перший елемент (29) як відсортований.

Потім він вибирає другий елемент (10), який є в невідсортованому списку, і порівнює його з попереднім, поміщеним у відсортований список (його частину). Оскільки 10 менше, ніж 29, він переміщує більший елемент праворуч і вставляє менший елемент на першу позицію. Тепер елементи 10 і 29 утворюють відсортований список.

Алгоритм виконує цю вправу послідовно, витягуючи елементи з несортованого списку праворуч і порівнюючи їх з елементами у відсортованому списку ліворуч, щоб визначити, в яке місце їх вставити.



Сортування вставками (*Insertion Sort*)

6 5 3 1 8 7 2 4

АЛГОРИТМ СОРТУВАННЯ ВСТАВКАМИ:

1. Запам'ятати в тимчасову змінну (buff) значення поточного елемента масиву;
2. Доки елементи ліворуч від нього більші за buff, переміщуємо їх на позицію праворуч. Тобто попередній елемент займе місце поточного. А той, що стоїть перед попереднім, переміститься в свою чергу на місце попереднього. І так елементи будуть рухатися один за одним.
3. Рух елементів закінчується, якщо черговий елемент, який потрібно зрушити, виявиться за значенням менше, ніж той, що запам'ятали в тимчасову змінну на початку циклу.
4. Цикл бере наступний елемент і знову зрушує все, розташоване перед ним, і великі за значенням.

0	1	2	3	4	5	6
1	2	4	5	3	7	6

0	1	2	3	4	5	6
1	2	4	3	5	7	6

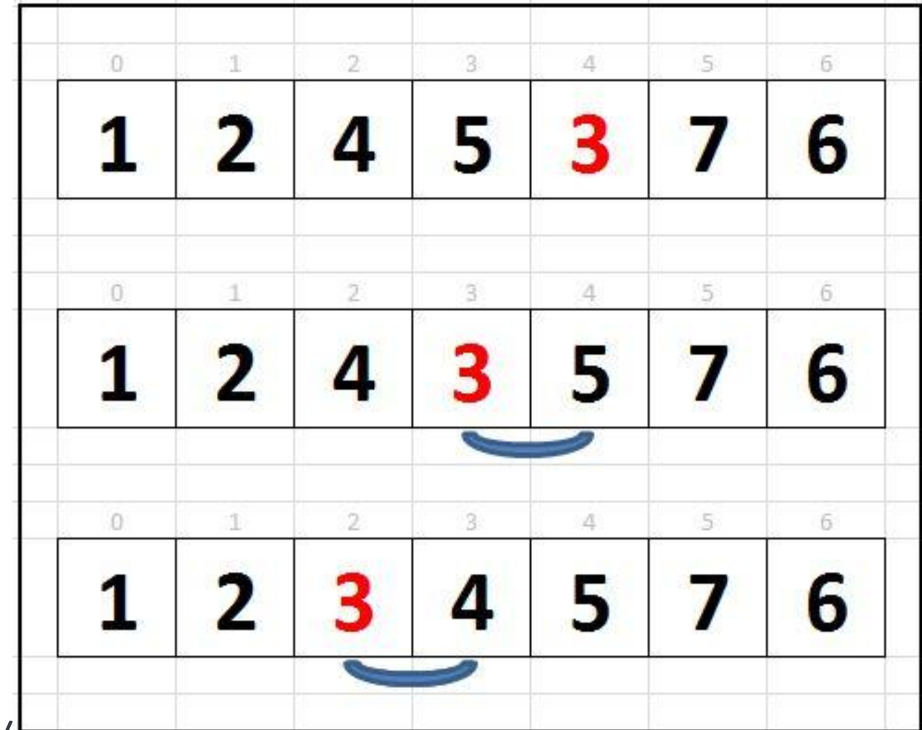
0	1	2	3	4	5	6
1	2	3	4	5	7	6

На першій ітерації в змінну-буфер запишеться значення з комірки з індексом 1 і цикл буде перевіряти цей елемент. Там у нас зараз число 2. Воно більше значення, яке записано в нульовій комірці, тому переміщення не буде.

Далі в змінну-буфер запишеться значення з комірки з індексом 2 і знову піде порівняння зі значеннями ліворуч і т.д.

Тільки на четвертій ітерації зовнішнього циклу почнеться перезапис значень. Трійка спочатку поміняється місцями з п'ятіркою, а потім з четвіркою.

Отже, в процесі **сортування вставками** елемент записаний в **buff** “просівається” до початку масиву. А у випадках, коли буде знайдений елемент зі значенням менший за **buff** або буде досягнуто початку послідовності – просіювання зупиняється



4	6	5	12	2	3	8	13	7	1	11	16	10	14	15	9
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

01:09

```
int main()
{
const int N = 10;
int a[N] = { 12, 5, 3, 2, 45, 96, 6, 8, 11, 24 };

int buff = 0; // для зберігання пересувного значення
for (int i = 1; i < N; i++)
{
   buff = a[i]; // запам’ятаємо пересувний елемент і почнемо перебір елементів ліворуч від нього
   for (int j = i - 1; j >= 0 && a[j] > buff; j--) // доки не виявиться елемент, менший за пересувний,
        a[j + 1] = a[j];
   a[j + 1] = buff; // поставимо пересувний елемент на його нове місце
}

for (int i = 0; i < N; i++) cout << a[i] << '\t';
}
```

Основний цикл алгоритму починається не з 0-го елемента, а з 1-го, оскільки елемент до 1-го елемента буде нашою відсортованою послідовністю (пам'ятаємо, що масив, що складається з одного елемента, є відсортованим) і вже щодо цього елемента з номером 0 ми будемо вставляти решту.

Власне код:

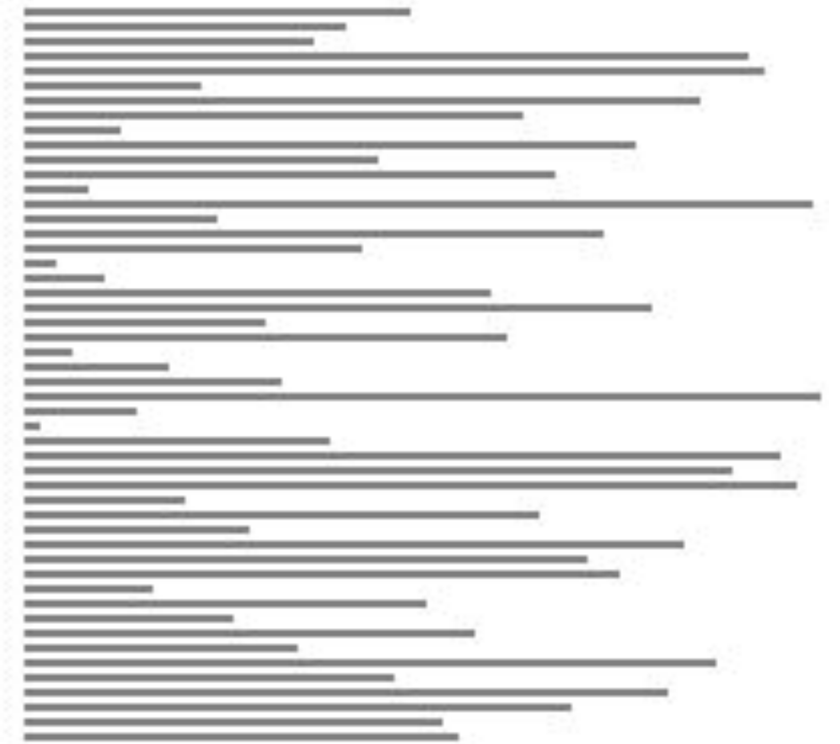
```
for(int i=1;i<n;i++)
  for(int j=i; j>0 && x[j-1]>x[j]; j--) // поки j>0 и елемент j-1 > j
    swap(x[j-1],x[j]); // міняємо місцями елементи j та j-1
```

Вкладений цикл шукає місце для вставки.
Рекомендую запам'ятати цей алгоритм, щоб у разі потреби написати сортування не ганьбитися сортуванням бульбашкою:)

Аналіз продуктивності

Сортування вставками має складність n^2
Кількість порівнянь обчислюється за формулою n*(n-1)/2.

При n=100 кількість перестановок дорівнює 4950, а не 10000, якби ми вираховували за формулою n^2.
Майте це на увазі при виборі алгоритму сортування.



Ефективність
Сортування вставкою є адаптивним, що означає, що воно зменшує загальну кількість кроків, якщо частково відсортований.

Сортування вставками найбільш ефективне, коли масив вже частково відсортований і коли елементів масиву небагато.

Якщо ж елементів менше 10, то даний алгоритм є найкращим.

Як і сортування вибором, сортування вставкою не підходить для великих обсягів даних, де він програє іншим алгоритмам сортування.

На прикладі простих вставок показово виглядає головна перевага більшості (але не всіх!) сортувань вставками, а саме дуже швидка обробка майже впорядкованих масивів.

Цьому сприяє сама основна ідея цього класу — перекидання елементів із невідсортованої частини масиву у відсортовану. При близькому розташуванні близьких за величиною даних місце вставки зазвичай знаходиться близько до краю відсортованої частини, що дозволяє вставляти з найменшими накладними витратами.

Немає нічого кращого для обробки майже впорядкованих масивів, ніж сортування вставками. Коли Ви десь зустрічаєте інформацію, що найкраща тимчасова складність сортування вставками дорівнює O(**n**) , то, швидше за все, маються на увазі ситуації з майже впорядкованими масивами.

Витрати часу на роботу даного алгоритму повністю залежать від початкових даних: кількості елементів в масиві і його початкової впорядкованості.

Це зрозуміло, що чим більше масив – тим більше часу треба на його обробку. Також більше часу буде потрібно на сортування масиву, в якому значення абсолютно не впорядковані.

Алгоритм **Сортування вставками** хороший для невеликих масивів (до декількох десятків елементів). Ще ефективніше працює, якщо дані такого масиву раніше були частково відсортовані.

Якщо в масив будуть додаватися нові дані (нові елементи), алгоритм зможе їх сортувати по мірі їх додавання (на відміну від сортування бульбашкою і сортування вибором).

Ефективність алгоритму значно зросте, якщо додати в код алгоритм бінарного пошуку.

Пáрне сортування простими вставками

Модифікація простих вставок розроблена в таємних лабораторіях корпорації Oracle. Це сортування входить до пакету JDK, є складовою частиною Dual-Pivot Quicksort.

Використовується для сортування малих масивів (до 47 елементів) та сортування невеликих ділянок великих масивів.

У буфер відправляються не один, а відразу два елементи, що стоять поруч. Спочатку вставляється більший елемент із пари і відразу після нього метод простої вставки застосовується до меншого елемента з пари.

7	9	1	5	11	3	2	6	4	13	14	10	12	8
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Що це дає? Економію для обробки меншого елемента із пари. Для нього пошук точки вставки та сама вставка здійснюються тільки на тій відсортованій частині масиву, в яку не входить відсортована область, яка задіяна для обробки більшого елемента з пари. Це стає можливим, оскільки більший та менший елементи обробляються відразу один за одним в одному проході зовнішнього циклу.

На середню складність за часом це не впливає (вона так і залишається рівною O($\mathbf{n}^2$)), проте парні вставки працюють трохи швидше, ніж звичайні.

```
void BinIns(int A[], int nn) {
  int I, j, x, m, L, R;
  for (i = 1; i < nn; i++) {
    x = A[i]; L = 0; R = i;
    while (L < R) {
      m = (L + R) / 2;
      if (A[m] <= x)
        L = m + 1;
      else R = m;
    }
  for (j = i; j >= R; j--)
    A[j] = A[j - 1];
  A[R] = x;
  }
}
```

7	9	1	5	11	3	2	6	4	13	14	10	12	8
1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
for (int k = left; ++left <= right; k = ++left)
{ //чергову пару поруч розташованих елементів заносимо
  int a1 = a[k], a2 = a[left]; //в пару буферних змінних
  if (a1 < a2) { a2 = a1; a1 = a[left]; } //Вставляємо більший елемент з пари
  while (a1 < a[--k]) { a[k + 2] = a[k]; }
  a[++k + 1] = a1; //Вставляємо менший елемент з пари
  while (a2 < a[--k]) { a[k + 1] = a[k]; }
  a[k + 1] = a2;
 }

//Граничний випадок, коли в масиві непарна кількість елементів
//Для останнього елемента застосовуємо сортування простими вставками
int last = a[right];
while (last < a[--right]) { a[right + 1] = a[right]; }
a[right + 1] = last;
```

Сортування простими вставками з бінарним пошуком

Позаяк місце для вставки шукається у відсортованій частині масиву, то ідея використати бінарний пошук напрошується сама собою. Інша справа, що пошук місця вставки не є критичним для тимчасової складності алгоритму (головний пожирач ресурсів - етап самої вставки елемента в знайдену позицію), тому оптимізація тут мало що дає.

А у випадку майже відсортованого масиву бінарний пошук може працювати навіть повільніше, оскільки він починає з середини відсортованої ділянки, яка, швидше за все, буде знаходитися надто далеко від точки вставки (а на звичайний перебір від позиції елемента до точки вставки піде менше кроків, якщо дані у масиві загалом упорядковані).

7	6	14	8	16	9	13	15	2	1	12	4	3	11	5	10
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

```
for  i:= 2  to  n  do
   if  a[i-1]>a[i]  then
   begin
      x:= a[i];
      left:= 1;
      right:= i-1;
      repeat
         sred:= (left+right) div 2;
         if  a[sred]<x  then
            left:= sred+1
            else  right:= sred-1;
      until  left>right;
      for j:=i-1 downto left  do
         a[j+1]:= a[j];
      a[left]:= x;
    end;
```

Тепер на кожному кроці виконується не ***N*** , а ***log 2 N*** перевірок, що вже значно краще (наприклад, порівняйте ***1000*** і ***10*** = ***log 2 1024***).
Отже, все буде скоєно ***N * l og 2 N*** порівнянь.
За кількістю пересилок алгоритм, як і раніше, має складність ***O(N 2)*** .

На захист бінарного пошуку зазначу, що він може сказати вирішальне слово в ефективності інших сортувань вставками. Завдяки йому, зокрема, на середню складність часу O(**n** log **n**) виходять такі алгоритми як сортування бібліотекара і пасьянсе сортування. Але про них згодом самостійно.

2. Сортування Шелла

Сортування Шелла — це алгоритм сортування, що є узагальненням сортування вставкою (включенням).

Алгоритм базується на двох тезах:

1) Сортування включенням ефективне для майже впорядкованих масивів.

2) Сортування вставкою не ефективне, оскільки переміщує елемент тільки на одну позицію за раз.

Тому сортування Шелла виконує декілька впорядкувань вставкою, кожен раз порівнюючи і переставляючи елементи, що розташовані на різній відстані один від одного.

Алгоритм сортування Шелла дозволяє уникати великих зрушень, як у випадку сортування вставкою, коли менше значення знаходиться наприкінці масиву і має бути переміщено в крайню ліву частину.

Ідея

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.

Цей алгоритм є доволі ефективним для наборів даних середніх розмірів. У найгіршому випадку він має складність O (n).

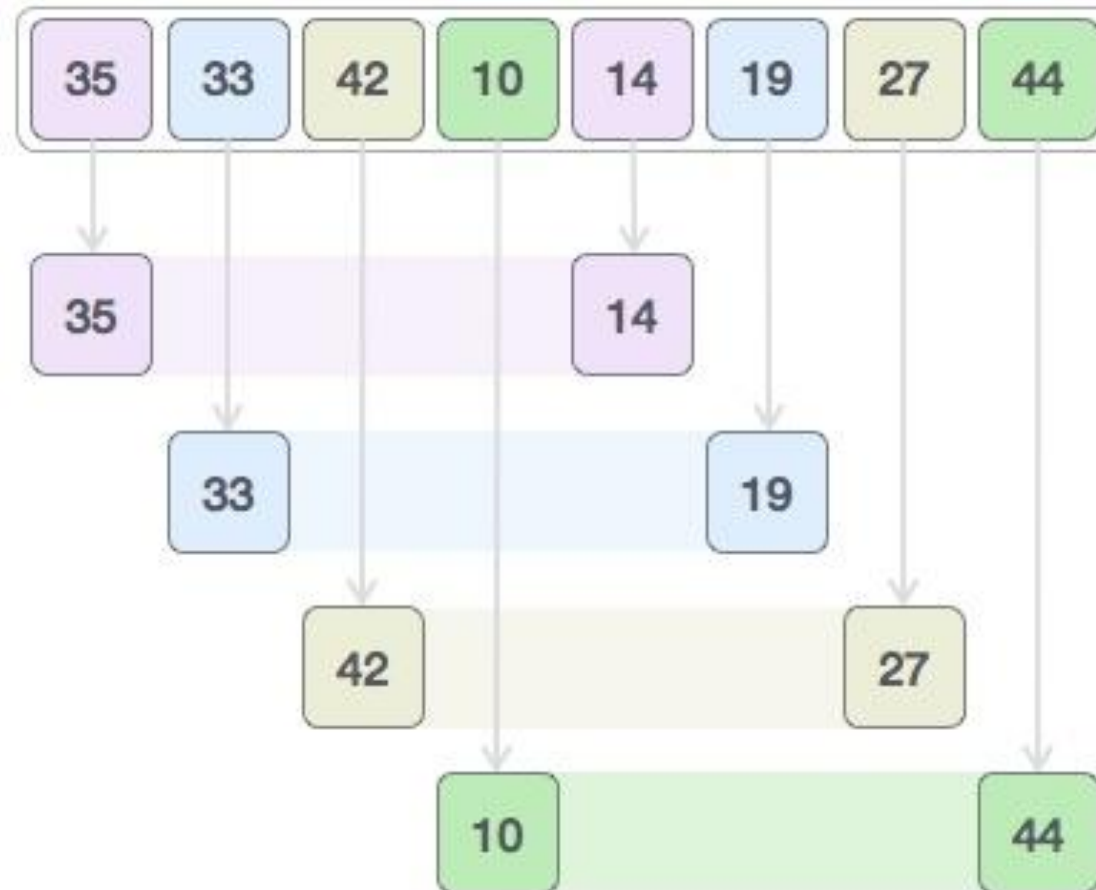
7	12	5	16	1	2	3	14	10	8	9	15	13	11	6	4
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

01:40

Як працює сортування Шелла?

Приклад: масив {35, 33, 42, 10, 14, 19, 27, 44}.

Для цього прикладу візьмемо інтервал 4. Зробімо віртуальний підсписок всіх значень, розташованих на відстані 4 позицій. Ось ці значення: {35, 14}, {33, 19}, {42, 27} і {10, 44}



comparison

> item to insert

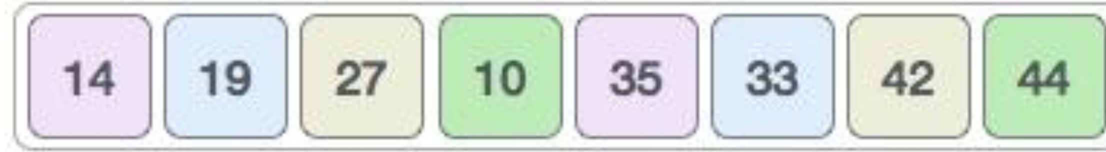
Shell Sort

gap: 5

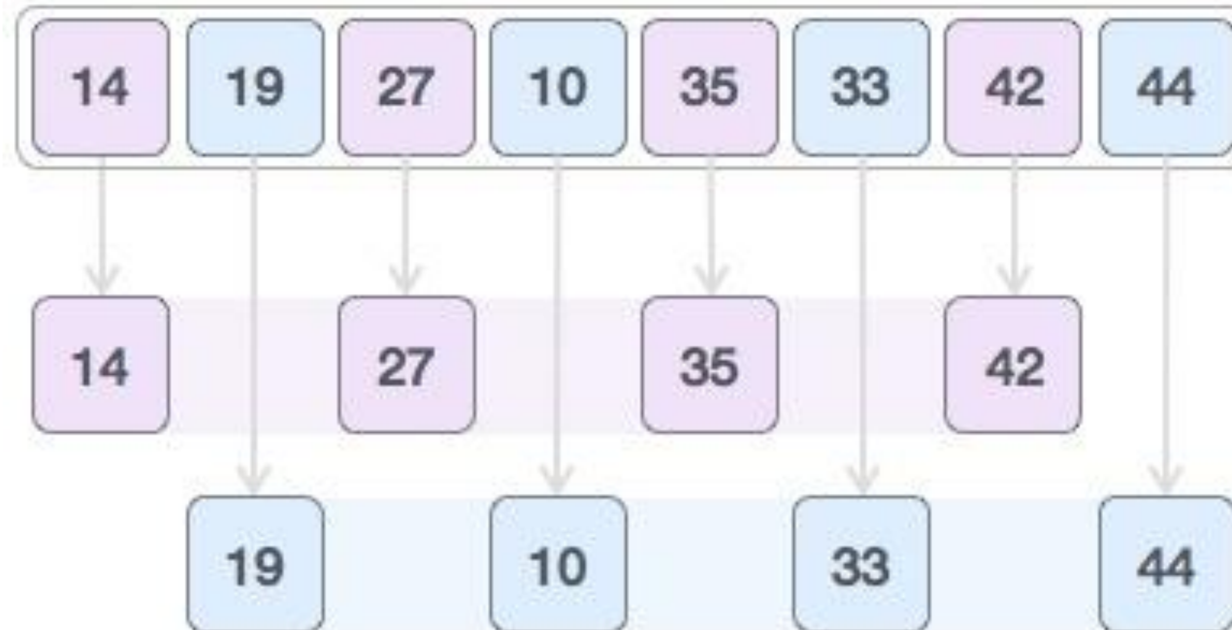
Sorts App

available on Android Market

Порівнюємо значення в кожному підсписку і переставляємо їх (якщо необхідно). Після цього кроку новий масив повинен виглядати наступним чином:



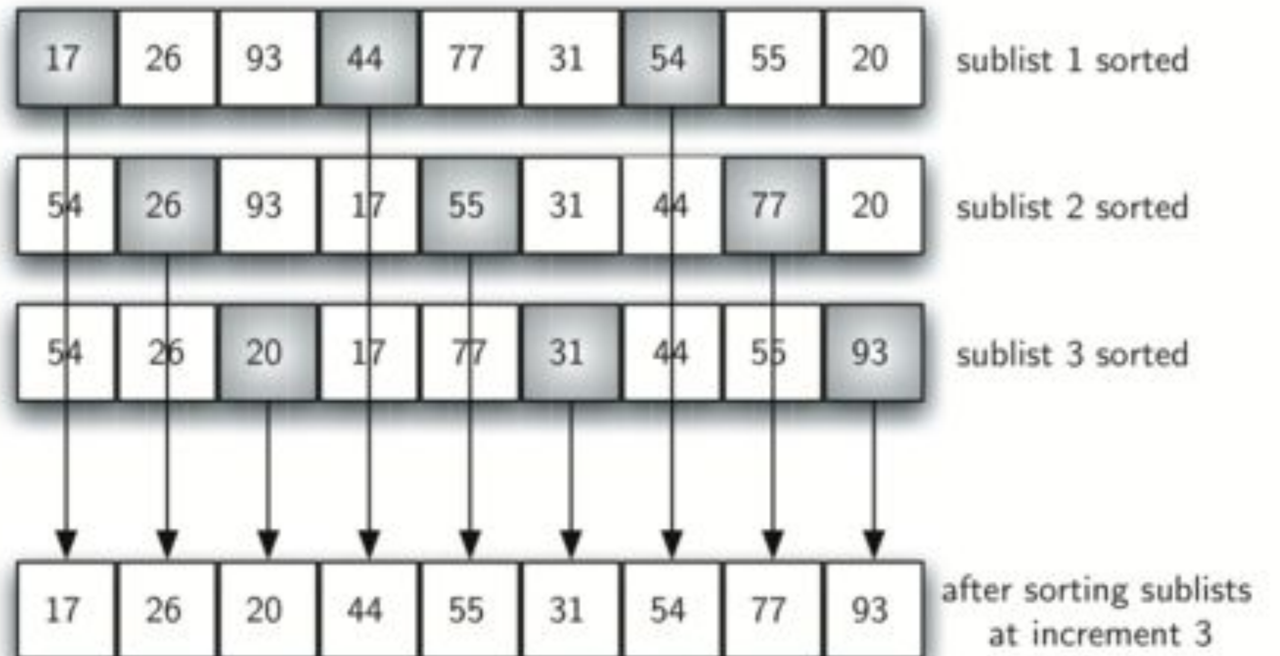
Потім ми беремо інтервал 2 і цей проміжок генерує два підсписки - {14, 27, 35, 42}, {19, 10, 33, 44}



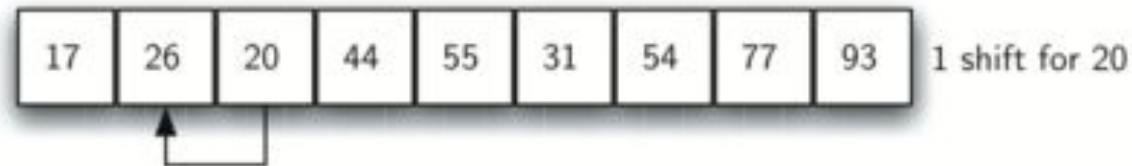
Порівнюємо і переставляємо значення, якщо потрібно. Далі сортуємо масив, використовуючи інтервал значення 1 алгоритмом сортування вставкою.

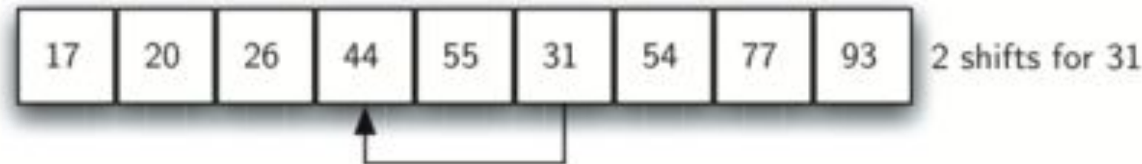
Приклад
Перший етап – сортування з кроком 3:

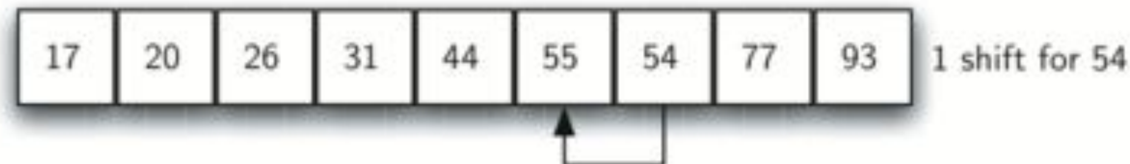
Ми починаємо з n / 2 підсписків. На наступному проході упорядковуємо n / 4 підсписків. Нарешті, єдиний список сортується за допомогою базового сортування вставками.

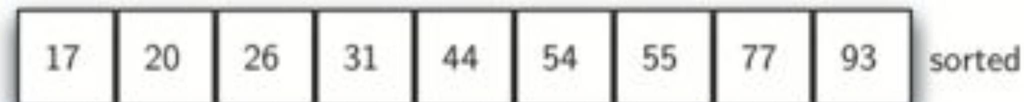


Другий етап – остаточне сортування вставками з використанням кроку 1 - іншими словами, стандартне сортування вставками.
Завдяки проведенню попередніх угруповань підсписків, тепер ми скоротили загальну кількість операцій зсуву, необхідних для розташування елементів списку в правильному порядку. У нашому випадку потрібно всього чотири переміщення, щоб завершити процес:









Алгоритм сортування Шелла

Крок 1 - ініціалізувати значення h

Крок 2 - Розділити список на менші підрозділи з рівним інтервалом h

Крок 3 - Відсортувати ці підсписки за допомогою сортування вставкою

Крок 4 - Повторювати, поки не буде відсортовано повний список

35	33	42	10	14	19	27	44

1st iteration	step=4
assignments	comparisons
3	1
3	1
3	1
	1

14	19	27	10	35	33	42	44	2nd iteration	step=2
14	19	27	10	35	33	42	44		
								assignments	comparisons
14		27		35		42		0	3
	19		10		33		44	3	1
	10		19		33		44	0	1
	10		19		33		44	0	1

								3rd iteration	step=1	insertion sort
14	10	27	19	35	33	42	44	3	1	
10	14	27	19	35	33	42	44	0	1	
10	14	27	19	35	33	42	44	3	2	
10	14	19	27	35	33	42	44	0	1	
10	14	19	27	35	33	42	44	3	2	
10	14	19	27	33	35	42	44	0	1	
10	14	19	27	33	35	42	44	0	1	

Comparisons	19
Assignments	21

Псевдокод

```
h = ПочатковийКрок;
поки h>0
{
   для i від h до n-1
   {
      x = a[i]; j = i - h;
      поки j>=0 і x < a[j]
      {
         a[j+h]=a[j]; j = j - h
      }
      a[j+h] = x
   }
   h=НовийКрок(h)
}
```

```
#Shell Sort Algorithm
def shellSort(data, length):
    gap = length//2
    while gap > 0:
      for iIndex in range(gap, length):
        temp = data[iIndex]
        jIndex = iIndex
        while jIndex >= gap and data[jIndex - gap] > temp:
          data[jIndex] = data[jIndex - gap]
          jIndex -= gap
        data[jIndex] = temp
      gap //= 2 print(data)
```