

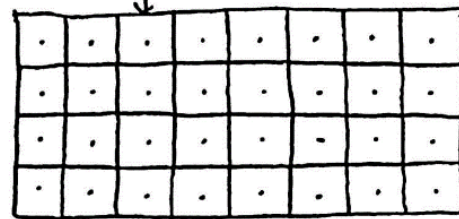
ЛЗ.Алгоритми сортування

1. Масиви і зв'язані списки

Уявіть, що ви прийшли в театр і хочете залишити свої особисті речі в гардеробі. Для зберігання речей є спеціальні ящики. У кожному ящику поміщається один предмет. Ви хочете здати на зберігання дві речі, тому вимагаєте виділити вам два ящика. По суті, саме так працює пам'ять вашого комп'ютера. Вона являє собою щось на зразок величезного шафи з безліччю ящиків, і у кожного ящика є адреса.



АДРЕС: fe0fffeb



Кожен раз, коли ви хочете зберегти в пам'яті окреме значення, ви запитуєте у комп'ютера місце в пам'яті, а він видає адреса для збереження значення. Якщо ж вам знадобиться зберегти кілька елементів, це можна зробити двома основними способами: скористатися масивом або списком.

Іноді в пам'яті потрібно зберегти список елементів. Припустимо, ви пишете додаток для управління поточними справами. Описи завдань повинні зберігатися у вигляді списку в пам'яті. Що використовувати – масив або пов'язаний список? Для початку спробуємо зберегти завдання в масиві, тому що цей спосіб більш зрозумілий. При використанні масиву всі завдання зберігаються в пам'яті безперервно (тобто поруч один з одним).

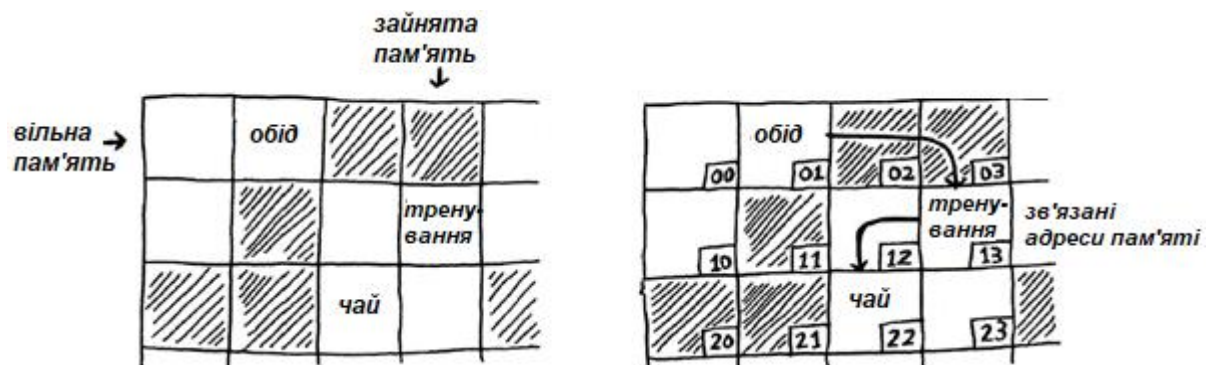


Тепер припустимо, що ви захотіли додати четверте завдання. Але наступний ящик вже зайнятий - там лежать чужі речі! Уявіть, що ви пішли в кіно з друзями і знайшли місця для своєї компанії, але тут приходить ще один друг, і йому сісти вже нікуди. Доводиться шукати нове місце, де зможуть розміститися всі. В цьому випадку вам доведеться запросити у комп'ютера інший блок пам'яті, в якому помістяться всі чотири завдання, а потім перемістити всі свої завдання туди.

Якщо раптом прийде ще один друг, місця знову не вистачить, і вам усім доведеться переміщатися знову! Крім того, додавання нових елементів в масив стане серйозною проблемою. Якщо вільного місця немає і вам кожен раз доводиться переміщатися в нову область в пам'яті, операція додавання нового елемента буде виконуватися дуже повільно. Найпростіше рішення – «бронювання місць»: навіть якщо список складається всього з 3 завдань, ви запитуєте у комп'ютера місце на 10 позицій просто про всяк випадок. Тоді в список можна буде додати до 10 завдань, і нічого переміщати не доведеться. Це непогане обхідний рішення, але у нього є пара недоліків:

- зайве місце може не знадобитися, і тоді пам'ять буде витрачатися неефективно. Ви її не використовуєте, однак ніхто інший її використовувати теж не може;
- якщо в список буде додано більше 10 завдань, переміщатися все одно доведеться.

При використанні пов'язаного списку елементи можуть розміщуватися де завгодно в пам'яті.



У кожному елементі зберігається адреса наступного елемента списку. Таким чином, набір довільних адрес пам'яті об'єднується в ланцюжок. Додати новий елемент в зв'язаний список простіше простого: просто розмістіть його за будь-якою адресою пам'яті і збережіть цю адресу в попередньому елементі. Зі зв'язаними списками нічого переміщати в пам'яті не потрібно. Також сама собою вирішується інша проблема: припустимо, ви прийшли в кіно з п'ятьма друзями. Ви намагаєтеся знайти місце на шістьох, але кінотеатр вже забитий, і знайти шість сусідніх місць неможливо. У зв'язаному списку ви фактично домовляєтесь: «Сідаємо на вільні місця і дивимося кіно». Якщо необхідне місце є в пам'яті, ви зможете зберегти дані в зв'язаному списку.

На сайтах з усілякими хіт-парадами і «першими десятками» застосовується шахрайська тактика для збільшення кількості переглядів. Замість того щоб вивести весь список на одній сторінці, вони розміщують по одному елементу на сторінці і змушують вас натискати кнопку Next для переходу до наступного елемента. Ви починаєте з № 10 і натискаєте Next на кожній сторінці, поки не дійдете до № 1. У результаті сайту вдається показати вам рекламу на цілих 10

сторінках, але натискати Next 9 раз для переходу до першого місця нудно. Було б набагато краще, якби весь список містився на одній сторінці, а ви б могли просто клацнути на імені людини для отримання додаткової інформації.

Схожа проблема існує і у зв'язаних списків. Припустимо, ви хочете отримати останній елемент зв'язаного списку. Просто прочитати потрібне значення не вдасться, тому що ви не знаєте, за якою адресою воно зберігається. Замість цього доведеться спочатку звернутися до елементу № 1 і дізнатися адресу елемента № 2, потім звернутися до елементу № 2 і дізнатися адресу елемента № 3 ". І так далі, поки не дійдете до останнього елемента. Зв'язані списки відмінно підходять в тих ситуаціях, коли дані повинні читатися послідовно: спочатку ви читаєте один елемент, за адресою переходите до наступного елемента і т. д. Але якщо ви маєте намір стрибати по списку туди-сюди, тримайтеся подалі від пов'язаних списків.

З масивами справа йде зовсім інакше. Працюючи з масивом, ви заздалегідь знаєте адресу кожного його елемента. Припустимо, масив містить п'ять елементів і ви знаєте, що він починається з адреси 00. За якою адресою зберігається п'ятий елемент?



Найпростіша математика дає відповідь: це адреса 04. Масиви прекрасно підходять для читання елементів в довільних позиціях, тому що звернення до будь-якого елемента в масиві відбувається миттєво. У пов'язаному списку елементи не зберігаються поруч один з одним, тому миттєво визначити позицію і-го елемента в пам'яті неможливо - потрібно звернутися до першого елемента, щоб отримати адресу другого елемента, потім звернутися до другого елемента для отримання адреси третього - і так далі, поки ви не дійдете до і-го.

Елементи масиву пронумеровані, причому нумерація починається з 0, а не з 1. Наприклад, в цьому масиві значення 20 знаходиться в позиції 1.

10	20	30	40
0	1	2	3

Позиція елемента називається його індексом. Таким чином, замість того щоб казати «Значення 20 знаходиться в позиції 1», правильно сказати «Значення 20 має індекс 1». Час виконання основних операцій з масивами і списками

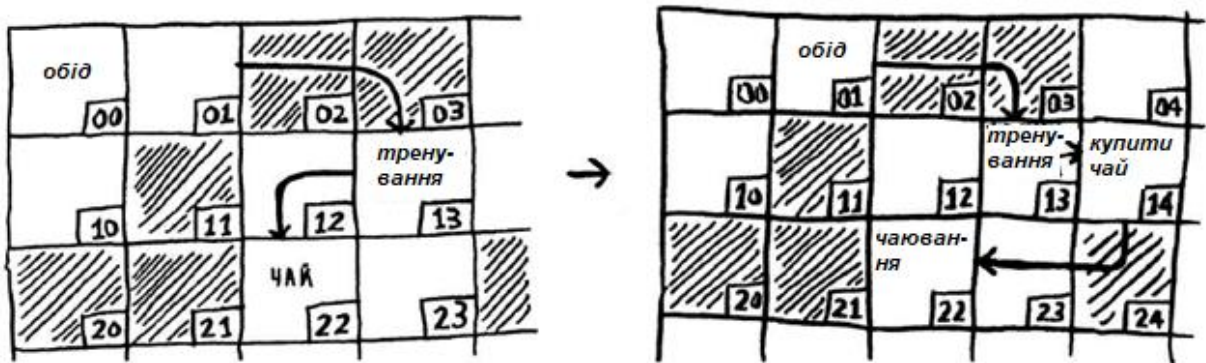
	масиви	списки	
читання	$O(1)$	$O(n)$	$O(n)$ - лінійний час $O(1)$ - постійний час
вставка	$O(n)$	$O(1)$	

Вставка в середину списку

Припустимо, ви вирішили, що список завдань повинен більше нагадувати календар. Перш дані додавалися тільки в кінець списку, а тепер вони повинні додаватися в порядку їх виконання.



Що краще підійде для вставки елементів в середину: масиви або списки? Зі списком завдання вирішується зміною покажчика в попередньому елементі.



А при роботі з масивом доведеться зрушувати вниз всі інші елементи.



А якщо вільного місця не залишилося, всі дані доведеться скопіювати в нову область пам'яті! Загалом, списки краще підходять для вставки елементів в середину.

Видалення

Що, якщо ви захочете видалити елемент? І знову список краще підходить для цієї операції, тому що в ньому досить змінити покажчик в попередньому елементі. У масиві при видаленні елемента всі наступні елементи потрібно буде зрушити вгору. На відміну від вставки видалення можливо завжди. Спроба вставки може бути невдалою, якщо в пам'яті не залишилося вільного місця. З

видаленням подібних проблем не буває. Час виконання основних операцій з масивами і зв'язаними списками.

	масиви	списки
читання	$O(1)$	$O(n)$
вставка	$O(n)$	$O(1)$
видалення	$O(n)$	$O(1)$

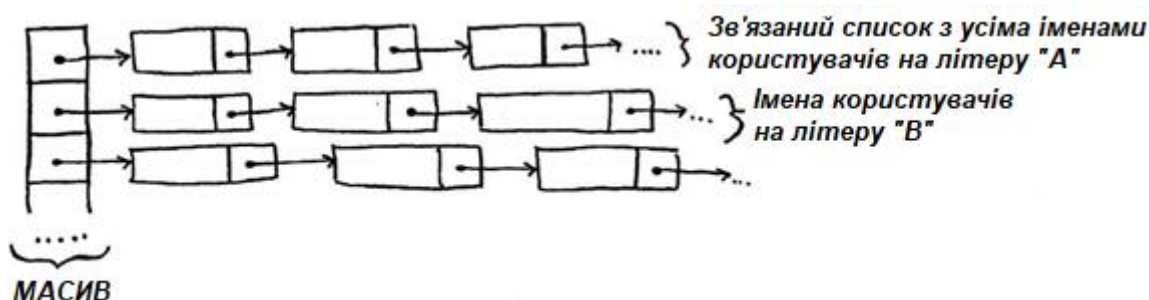
Масиви надзвичайно популярні через те, що вони підтримують довільний доступ. Масиви володіють більш високою швидкістю читання; це пояснюється тим, що вони підтримують довільний доступ. Багато реальні ситуації вимагають довільного доступу, тому масиви часто застосовуються на практиці. Також масиви і списки використовуються для реалізації інших структур даних.

Вправи

- Припустимо, Facebook зберігає список імен користувачів. Коли хтось намагається зайти на сайт Facebook, система намагається знайти своє ім'я користувача. Якщо ім'я входить в список імен зареєстрованих користувачів, то вхід дозволяється. Користувачі приходять на Facebook досить часто, тому пошук за списком імен користувачів буде виконуватися часто. Будемо вважати, що Facebook використовує бінарний пошук для пошуку в списку. Бінарному пошуку необхідний довільний доступ – алгоритм повинен миттєво звернутися до середнього елементу поточної частини списку. Знаючи цю обставину, як би ви реалізували список користувачів: у вигляді масиву або у вигляді зв'язаного списку?

- Користувачі також досить часто створюють нові облікові записи на Facebook. Припустимо, ви вирішили використовувати масив для зберігання списку користувачів. Якими недоліками володіє масив для виконання вставки? Припустимо, ви використовуєте бінарний пошук для знаходження облікових даних. Що станеться при додаванні нових користувачів в масив?

- У дійсності Facebook не використовує ні масив, ні зв'язаний список для зберігання інформації про користувачів. Розглянемо гібридну структуру даних: масив зв'язаних списків. Є масив з 26 елементів. Кожен елемент містить посилання на зв'язаний список. Наприклад, перший елемент масиву вказує на зв'язаний список всіх імен користувачів, що починаються на букву «А». Другий елемент вказує на зв'язаний список всіх імен користувачів, що починаються на букву «В», і т. Д.



Припустимо, користувач з ім'ям Adit - реєструється на Facebook і ви хочете додати його в список. Ви звертаєтесь до елементу 1 масиву, знаходите зв'язаний список елемента 1 і додаєте Adit в кінець списку. Тепер припустимо, що

zareєструвати потрібно користувача Zakhir. Ви звертаєтесь до елемента 26, який містить зв'язаний список всіх імен, що починаються з Z, і перевіряєте, чи присутній Zakhir в цьому списку.

Сортування вибором

Припустимо, у вас на комп'ютері записана музика і для кожного виконавця зберігається лічильник відтворення.

~♪~	Список відтворення
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

Ви хочете відсортувати список за спаданням лічильника відтворення.

Як це зробити? Одне з можливих рішень - пройти по списку і знайти виконавця з найбільшою кількістю відтворення. Цей виконавець додається в новий список.

~♪~	Список відтворення	→	~♪~	Список відтворення
RADIOHEAD	156		RADIOHEAD	156
KISHORE KUMAR	141			
THE BLACK KEYS	35			
NEUTRAL MILK HOTEL	94			
BECK	88			
THE STROKES	61			
WILCO	111			

1. У RADIOHEAD
НАЙБІЛЬША КІЛЬКІСТЬ
ВІДТВОРЕННЯ


2. ДОДАЄМО
RADIOHEAD ДО
НОВОГО СПИСКУ

Потім те ж саме відбувається з наступним за кількістю відтворення виконавцем.

~♪~	Список відтворення	→	~♪~	Список відтворення
RADIOHEAD	156		RADIOHEAD	156
KISHORE KUMAR	141		KISHORE KUMAR	
THE BLACK KEYS	35			
NEUTRAL MILK HOTEL	94			
BECK	88			
THE STROKES	61			
WILCO	111			

А тепер спробуємо оцінити те, що відбувається з точки зору теорії обчислень і подивимося, скільки часу будуть займати операції. Нагадаємо, що час $O(n)$

означає, що ви по одному разу звертаєтесь до кожного елементу списку. Наприклад, при: простому пошуку за списком виконавців кожен виконавець буде перевірений один раз.

1. RADIOHEAD
 2. KISHORE KUMAR
 3. THE BLACK KEYS
 4. NEUTRAL MILK HOTEL
 5. BECK
 6. THE STROKES
 7. WILCO
- 

Щоб знайти виконавця з найбільшим значенням лічильника відтворення, необхідно перевірити кожен елемент в списку. Це робиться за час $O(n)$. Отже, є операція, яка виконується за час $O(n)$, і її необхідно виконати n раз. Все це вимагає часу $O(n \times n)$, або $O(n^2)$. Реальний час $\frac{1}{2} n^2$

Алгоритми сортування дуже корисні. Наприклад, тепер ви можете відсортувати: імена в телефонній книзі; дати подорожей; повідомлення електронної пошти (від нових до старих).

Алгоритм сортування вибором легко пояснюється, але повільно працює. Швидке сортування – ефективний алгоритм сортування, який виконується за час $O(n \log n)$.

Приклад коду

Код виконує сортування масиву за зростанням.

```
def find Smallest(arr):
    smallest = arr[0]// Для зберігання найменшого значення
    smallest_index = 0 //Для зберігання індексу найменшого значення
    for i in range(1, len(arr)):
        if arr[i] < smallest:
            smallest = arr[i]
            smallest_index = i
    return smallest_index
```

На основі цієї функції можна написати функцію сортування вибором:

```
def selectionSort(arr):
    newArr = []
    for i in range(len(arr)):
        // Знаходимо мін. елемент у масиві та додаємо його в новий масив
        smallest = indSmallest(arr)
        newArr.append(arr.pop(smallest))
    return newArr

print selectionSort([5, 3, 6, 2, 10])
```

Висновки

- Пам'ять комп'ютера нагадує величезну шафу з ящиками.
- Якщо потрібно зберегти набір елементів, скористайтеся масивом або списком.
- У масиві всі елементи зберігаються в пам'яті поруч один з одним.
- У списку елементи розподіляються в довільних місцях пам'яті, при цьому в одному елементі зберігається адреса наступного елементу.
- Масиви забезпечують швидке читання.
- Списки забезпечують швидку вставку і виконання.
- Всі елементи масиву повинні бути однотипними (тільки цілі числа, тільки дійсні числа).

2. Алгоритми сортування

Завданням сортування є перетворення вихідного масиву в масив, що містить ті ж елементи, але в порядку зростання (або зменшення) значень. Методи, що сортують масив «на місці», діляться на три основні класи в залежності від способу, що лежить в їх основі.

1. Сортування включеннями (вставкою).
2. Сортування вибором.
3. Сортування обміном.

Сортування включеннями (вставкою)

1) Сортування простою вставкою

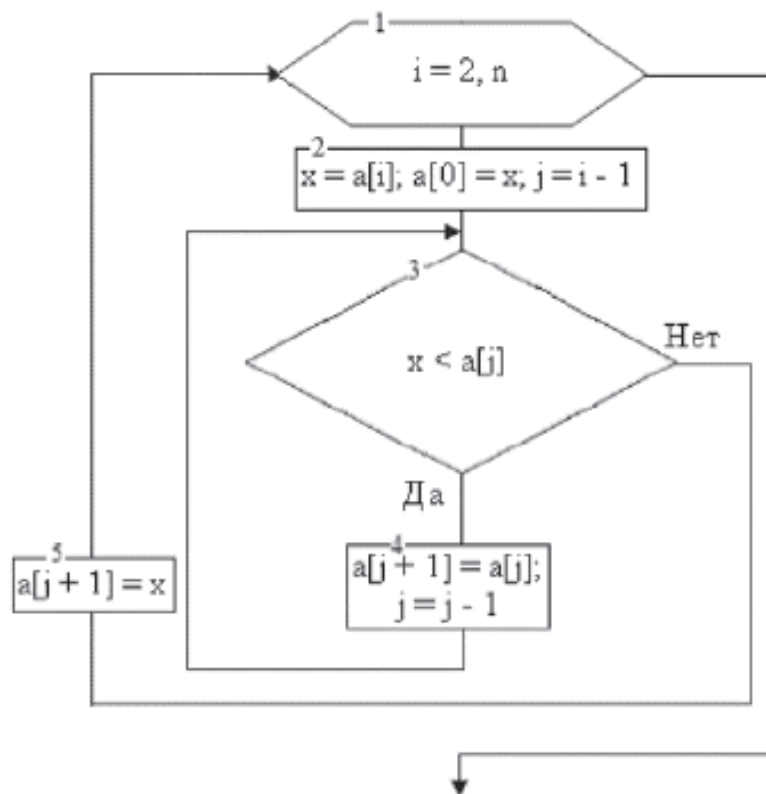
Нехай є масив $a[1], a[2], \dots, a[n]$. Нехай елементи $a[1], a[2], \dots, a[i-1]$ вже відсортовані, і нехай маємо вхідну послідовність $a[i], a[i+1], \dots, a[n]$. На кожному кроці, починаючи з $i = 2$ і збільшуючи i на одиницю, беремо i -й елемент вхідної послідовності і вставляємо його на відповідне місце в уже відсортовану частину послідовності. Позначимо елемент, що вставляється, через x . Нехай початковий масив 32 64 9 30 87 14 2 76.

- | | |
|----------------------|--|
| $i = 2 \quad x = 64$ | Шукаємо для x підходяще місце, вважаючи, що $a[1] = 32$ - це вже відсортована частина послідовності
32 64 9 30 87 14 2 76. |
| $i = 3 \quad x = 9$ | Частина послідовності $a[1], a[2]$ вже відсортована.
Шукаємо для x підходяще місце: 9 32 63 30 87 14 2 76. |
| $i = 4 \quad x = 30$ | Шукаємо для x підходяще місце: 9 30 32 64 87 14 2 76 |
| $i = 5 \quad x = 87$ | Шукаємо для x підходяще місце: 9 30 32 64 87 14 2 76. |
| $i = 6 \quad x = 14$ | Шукаємо для x підходяще місце: 9 14 30 32 64 87 2 76 |
| $i = 7 \quad x = 2$ | Шукаємо для x підходяще місце: 2 9 14 30 32 64 87 76 |
| $i = 8 \quad x = 76$ | Шукаємо для x підходяще місце: 2 9 14 30 32 64 76 87 |

При пошуку відповідного місця для елемента x чергували порівняння і пересилання. Як би «просівали» x , порівнюючи його з черговим елементом $a[i]$ і або вставляючи x , або пересилаючи $a[i]$ направо і просуваючись вліво. Зауважимо, що «просівання» може закінчитися при двох різних умовах:

- 1) знайдений елемент, значення якого більше, ніж x ;
- 2) досягнутий кінець послідовності.

Це типовий приклад циклу з двома умовами закінчення. При реалізації алгоритму з метою закінчення виконання внутрішнього циклу використовують прийом фіктивного елемента (бар'єру). Його можна встановити, прийнявши $a[0] = x$. Наведемо фрагмент блок-схеми алгоритму та її опис.



Блок 1. Починаємо цикл для перебору всіх елементів вихідного масиву.

Блок 2. Надаємо x значення чергового елемента масиву. Встановлюємо бар'єр і просуваємося назад по відсортованій частині масиву.

Блок 3. Починаючи цикл для пошуку відповідного місця для значення x .

Блок 4. Поки x менше чергового елемента (вихід «Так» блоку 3), виконуємо зсув та просуваємося до початку відсортованої частини масиву.

Блок 5. По виходу «Ні» блоку 3 вставляємо елемент x на потрібне місце.

Покажемо виконання алгоритму для розглянутого прикладу.

i	x	$a[0]$	j	$x < a[j]?$	$a[j + 1]$	Масив
2	64	64	1	$64 < 32?$ «Нет»	$a[2] = x = 64$	32 64 9 30 87 14 2 76
3	9	9	2	$9 < 64?$ «Да»	$a[3] = a[2] = 64$	32 64 64 30 87 14 2 76
			1	$9 < 32?$ «Да»	$a[2] = a[1] = 32$	32 32 64 30 87 14 2 76
			0	$9 < 9?$ «Нет»	$a[1] = x = 9$	9 32 64 30 87 14 2 76
4	30	30	3	$30 < 64?$ «Да»	$a[4] = a[3] = 64$	9 32 64 64 87 14 2 76
			2	$30 < 32?$ «Да»	$a[3] = a[2] = 32$	9 32 32 64 87 14 2 76
			1	$30 < 9?$ «Нет»		$a[2] = x = 30$

Після цього кроку масив виглядає так: 9 30 32 64 87 14 2 76. Сорткування триває далі для решти значень i .

2) Сорткування бінарними включеннями

Алгоритм сорткування простими включеннями має слабкі місця. Це, по-перше, необхідність переміщення даних, причому при вставці елементів, близьких до кінця масиву, доводиться переміщати майже весь масив. Другий недолік – це необхідність пошуку місця для вставки, на що також витрачається багато ресурсів. Цю частину алгоритму можна поліпшити, застосувавши так званий бінарний пошук. Цей метод на кожному кроці порівнює x із середнім елементом відсортованої послідовності доти, поки не буде знайдено місце включення. Модифікований алгоритм називається сорткуванням бінарними включеннями. Позначимо $left$ - ліва межа відсортованого масиву, $right$ - його права межа. Наведемо фрагмент блок-схеми алгоритму та її опис.

Блок 1. Починаємо цикл для перебору всіх елементів вихідного масиву.

Блок 2. Надаємо x значення чергового елемента масиву. Встановлюємо ліву і праву межу відсортованої частини масиву.

Блок 3. Поки ліва межа відсортованого масиву не перевищує праву, виконується тіло циклу, що складається з блоків 4-7.

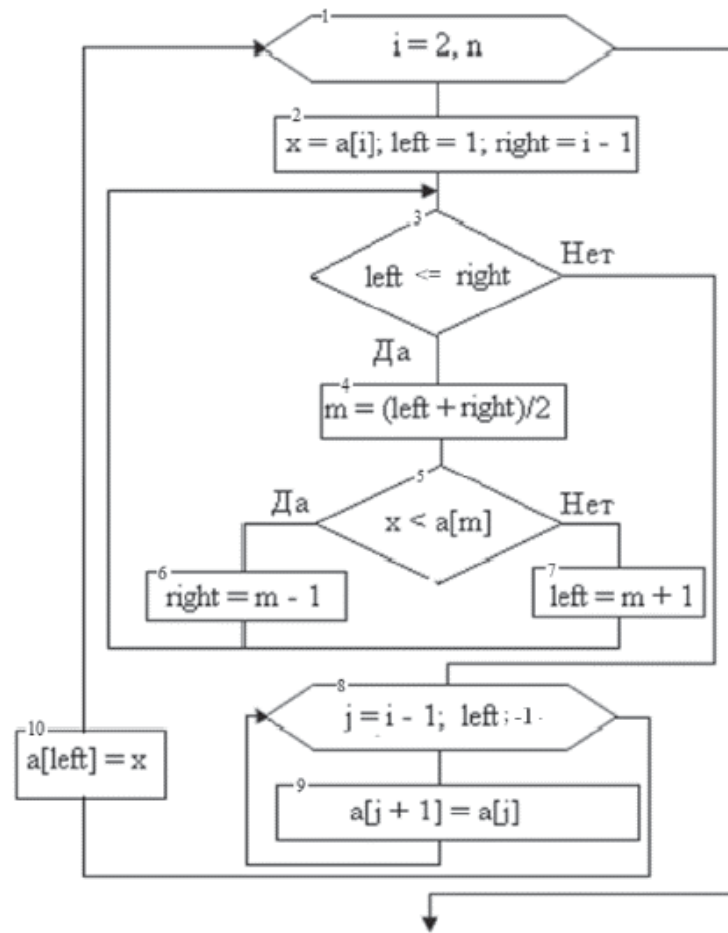
Блок 4. Знаходиться середній за положенням елемент відсортованої частини масиву.

Блок 5. Значення x порівнюється зі знайденим елементом. По виходу «Так» блоку 5 коригується права межа відсортованої частини масиву, по виходу «Ні» - ліва. При виході з циклу з передумовою буде знайдено положення елементу, що вставляється.

Блоки 8-9 реалізують зсув елементів масиву для вставки значення x .

Блок 10. Вставка значення в відсортовану частину масиву.

Наведемо виконання алгоритму при сортванні масиву 32 64 9 30 87 14 2 76.



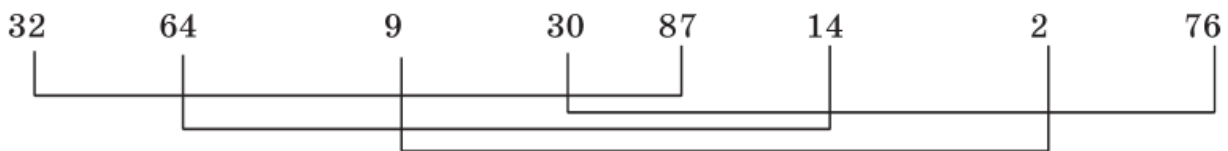
i	x	$left$	$right$	$left \leq right?$	m	$x < a[m]?$	j	$a[j + 1]$	$a[left]$
2	64	1	1	«Да»	1	32 < 64? «Да»			
			0	«Нет»			1	$a[2] = 64$	$a[1] = 32$
3	9	1	2	«Да»	1	9 < 64? «Да»			
			1	«Нет»			2	$a[3] = a[2] = 64$	
							1	$a[2] = a[1] = 32$	
									$a[1] = 9$
4	30	1	3	«Да»	2	30 < 32? «Да»			
			2	«Да»	1	30 < 9? «Нет»			
		2		«Нет»			3	$a[4] = a[3] = 64$	
							2	$a[3] = a[2] = 32$	$a[2] = 30$

Після цього кроку масив виглядає так: 9 30 32 64 87 14 2 76. Сортування масиву триває для наступних значень i .

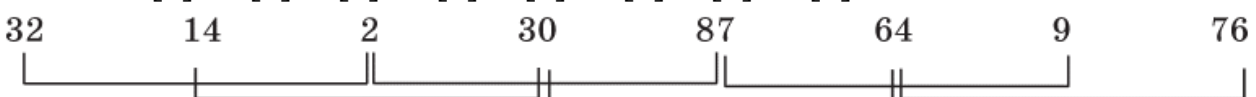
3) Сорткування Шела

Подальшим розвитком методу сорткування включеннями є сорткування методом Шела, яке ще називається сорткуванням включеннями з зменшуваним відстанню. Ми не будемо описувати алгоритм в загальному вигляді, обмежимося випадком, коли число елементів в сортованому масиві є ступенем числа 2. Для масиву з 2^n елементами алгоритм працює наступним чином. На першому етапі проводиться сорткування включенням всіх пар елементів масиву, відстань між якими дорівнює 2^{n-1} . На другому етапі проводиться сорткування включенням елементів отриманого масиву, відстань між якими дорівнює 2^{n-2} . Алгоритм триває до тих пір, поки ми не дійдемо до етапу з відстанню між елементами, що дорівнює одиниці, і не виконаємо завершальну сорткування включеннями. Покажемо сорткування Шела на масиві, що складається з 8 елементів. На першому проході окремо групуються і сортуються всі елементи, віддалені один від одного на чотири позиції. Цей процес називається 4-сорткуванням. У прикладі з восьми елементів кожна група містить рівно два елементи. Після цього елементи знову об'єднуються в групи з елементами, що відстоять один від одного на дві позиції, і упорядковано заново. Цей процес називається 2-сорткуванням. Нарешті, на третьому проході всі елементи упорядковано звичайної сорткуванням, або 1-сорткуванням. Початковий стан 32 64 9 30 87 14 2 76.

$n = 3$.



На першому кроці порівнюються елементи, віддалені один від одного на 4 позиції: $a[1]$ і $a[5]$, $a[2]$ і $a[6]$, $a[3]$ і $a[7]$, $a[4]$ і $a[8]$.



На другому кроці порівнюються елементи, віддалені один від одного на 2 позиції: $a[1]$ і $a[3]$, $a[2]$ і $a[4]$, $a[3]$ і $a[5]$, $a[4]$ і $a[6]$, $a[5]$ і $a[7]$, $a[6]$ і $a[8]$.



На цьому кроці порівнюються всі сусідні елементи. Відсортований масив: 2 9 14 30 32 64 76 87.

Сорткування включеннями виявляється не дуже хорошим методом для ПК, так як включення елемента з подальшим зсувом всього ряду елементів на одну позицію неекономна

Сортування простим вибором

У неупорядковому масиві вибирається і відділяється від інших елементів найменший елемент. Найменший елемент записується на i -е місце вихідного масиву, а елемент з i -го місця – на місце вибраного. Уже впорядковані елементи (а вони будуть розташовані починаючи з першого місця) виключаються з подальшого сортування, тому довжина неупорядкованого масиву, що залишився, повинна бути на один елемент менше попереднього. Сортування простим вибором продемонструємо на масиві 32 64 9 30 87 14 2 76.

$i = 1$, найменше значення 2, міняємо місцями 2 і 32. Масив після цього кроку:
2 64 9 30 87 14 32 76.

$i = 2$, найменше значення масиву 9, міняємо місцями 9 і 64. Масив після цього кроку:

2 9 64 30 87 14 32 76.

$i = 3$, найменше значення масиву 14, міняємо місцями 14 і 64. Масив після цього кроку:

2 9 14 30 87 64 32 76.

$i = 4$, найменше значення масиву 30. Обмін не відбувається, так як 30 знаходиться на своєму місці.

$i = 5$, найменше значення масиву 32, міняємо місцями 32 і 87. Масив після цього кроку:

2 9 14 30 32 64 87 76.

$i = 6$, найменше значення масиву 64. Обмін не відбувається, так як 64 знаходиться на своєму місці.

$i = 7$, найменше значення масиву 76, міняємо місцями 76 і 87. Масив після цього кроку:

2 9 14 30 32 64 76 87.

Масив відсортований, але виконується ще один крок.

$i = 8$, найменше значення масиву 87. Обмін не відбувається, так як 87 знаходиться на своєму місці.

Таким чином, метод сортування простим вибором заснований на повторному виборі найменшого елемента спочатку серед n елементів, потім серед $(n - 1)$ -го елемента і т. д. Можлива ситуація, коли обмін значень елементів не відбувається. Наведемо блок-схему і опис алгоритму сортування простим вибором.

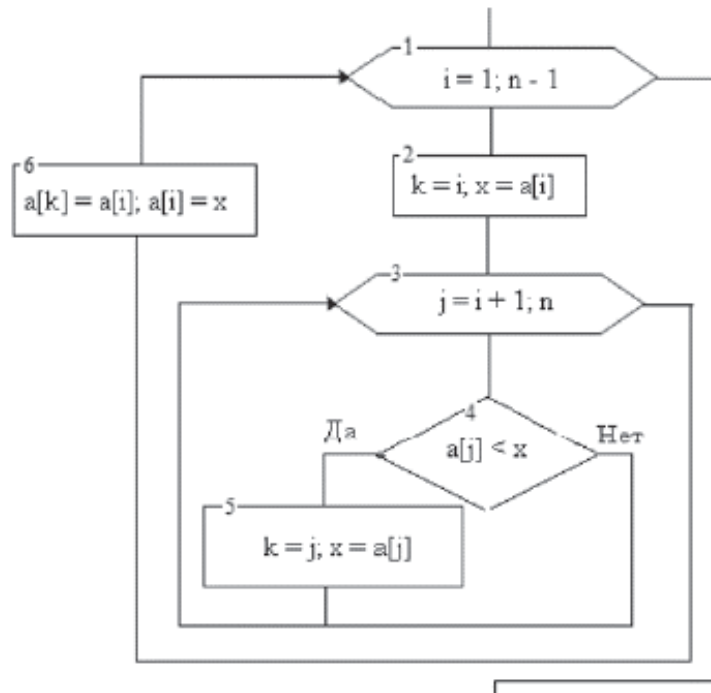
Блок 1. Організовує зовнішній цикл для перегляду елементів неупорядкованої частини масиву.

Блок 2. Запам'ятовування індексу і значення i -го елемента, які приймаємо за початкові при пошуку найменшого елемента.

Блоки 3-5. Пошук значення найменшого елемента і його номера в ще неупорядкованій частини масиву.

Блок 6. Обмін найменшого і i -го елементів. Наведемо виконання алгоритму для масиву:

32 64 9 30 87 14 2 76; $n = 8$.



i	k	x	j	$a[j] < x?$	$a[k] = a[i]; a[i] = x$
1	1	32	2	$a[2] < 32?$ «Ні»	
	3	9	3	$a[3] < 32?$ «Да»	
			4	$a[4] < 9?$ «Ні»	
			5	$a[5] < 9?$ «Ні»	
			6	$a[6] < 9?$ «Ні»	
	7	2	7	$a[7] < 9?$ «Да»	$A[7] = a[1] = 32; a[1] = 2$
			КЦ		

Після закінчення внутрішнього циклу масив має вигляд 2 64 9 30 87 14 32 76. Сортювання триває для всіх значень параметра зовнішнього циклу i , що залишився

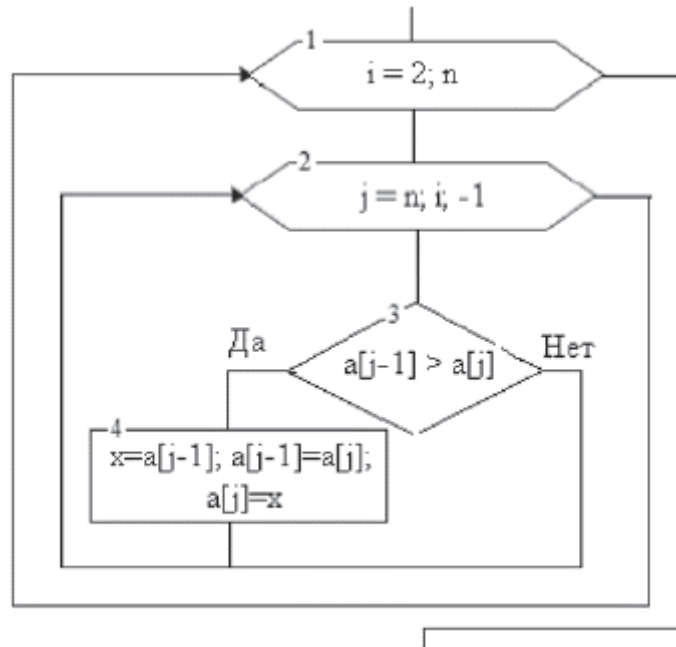
Обмін сортювання

1) Сортювання простим обміном

Простий обмін сортювання («метод бульбашки») для масиву $a[1], a[2], \dots, a[n]$ працює так. Починаючи з кінця масиву, порівнюємо два сусідні елементи ($a[n]$ і $a[n - 1]$). Якщо виконується умова $a[n - 1] > a[n]$, то значення елементів міняються місцями. Процес триває для $a[n - 1]$ і $a[n - 2]$ і т. д., поки не буде

виконано порівняння $a[2]$ і $a[1]$. Зрозуміло, що після цього на місці $a[1]$ виявиться елемент масиву з найменшим значенням. На другому кроці процес повторюється, але останніми порівнюються $a[3]$ і $a[2]$. На останньому кроці будуть порівнюватися тільки значення $a[n]$ і $a[n - 1]$. Зрозуміла аналогія з бульбашкою, оскільки найменші елементи (самі «легкі») поступово «спливають» до верхньої межі масиву.

Проста реалізація алгоритму.



Блок 1. Арифметичний цикл. Значення параметра циклу i визначає кількість елементів для порівняння у внутрішньому циклі.

Блок 2. Завдання номерів елементів для порівняння.

Блок 3. Порівняння двох сусідніх елементів.

Блок 4. Виконується обмін елементів масиву при виході «Так» блоку 3.

Сортуємо масив 32 64 9 30 87 14 2 76. Виконання алгоритму для $i = 2$ наведено нижче.

i	j	$a[j - 1] > a[j]?$	$a[j - 1], a[j]$	Масив
2	8	$2 > 76?$ «Нет»		32 64 9 30 87 14 2 76
	7	$14 > 2?$ «Да»	$a[6] = 2, a[7] = 14$	32 64 9 30 87 2 14 76
	6	$87 > 2?$ «Да»	$a[5] = 2, a[6] = 87$	32 64 9 30 2 87 14 76
	5	$30 > 2?$ «Да»	$a[4] = 2, a[5] = 30$	32 64 9 2 30 87 14 76
	4	$9 > 2?$ «Да»	$a[3] = 9, a[4] = 2$	32 64 2 9 30 87 14 76
	3	$64 > 2?$ «Да»	$a[2] = 2, a[3] = 64$	32 2 64 9 30 87 14 76
	2	$32 > 2?$ «Да»	$a[1] = 2, a[2] = 32$	2 32 64 9 30 87 14 76

Після цього кроку значення 2 встало на своє місце, і алгоритм повторюється для $i = 3, 4, \dots, n$.

Метод бульбашки працює нерівномірно для так званих «легких» та «важких» значень. Одна неправильно розташована «бульбашка» в «важкому» кінці розсортованого масиву впливе на місце за один прохід, а неправильно розташований елемент в «легкому» кінці буде опускатися на правильне місце тільки за один крок на кожному проході. Наприклад, масив

12 18 42 44 55 67 94 6

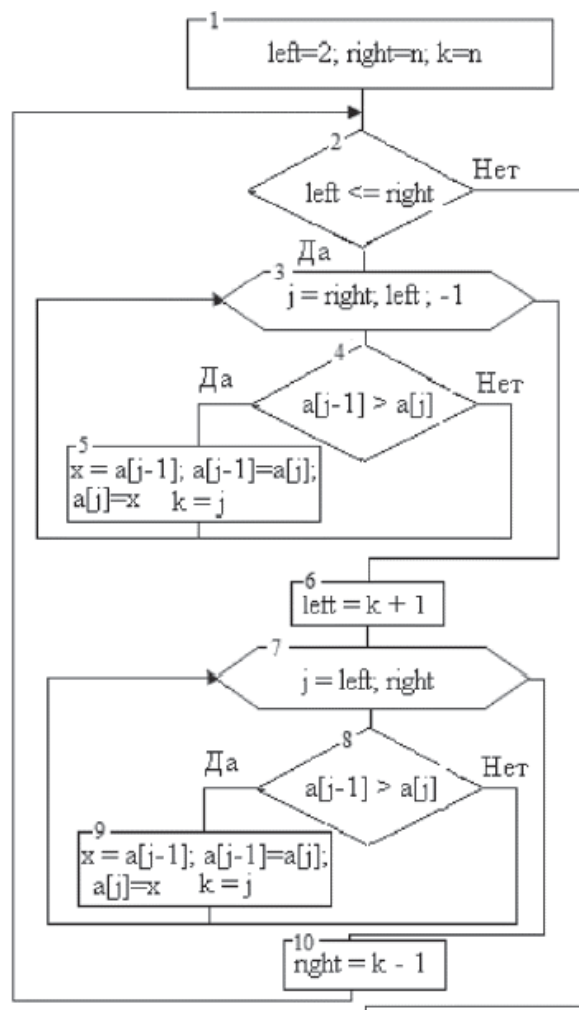
буде розсортований за один прохід, а сортування масиву

94 6 12 18 43 44 55 76

потребує семи проходів. Ця асиметрія підказує поліпшення: міняти місцями напрямком наступних один за іншим проходів. Цей покращений алгоритм називається шейкер-сортуванням.

2) Шейкер-сортування

При його застосуванні на кожному наступному кроці змінюється напрямок послідовного перегляду. В результаті на одному кроці «спливає» черговий найбільш легкий елемент, а на іншому «тоне» черговий найважчий. Позначимо $left$ - номер лівої межі сортованої частини масиву, $right$ - номер його правої межі.



Блок 1. Установка номерів початкових меж сортованого масиву.

Блок 2. Вхід в цикл. Поки ліва межа не перевищує праву межу (вихід «Так» блоку 2) виконуємо цикл.

Блок 3. Виконання проходу масиву вниз.

Блок 4. Порівняння сусідніх елементів.

Блок 5. Якщо $a[j - 1] > a[j]$ (вихід «Так» блоку 4), проводимо обмін цих елементів та фіксуємо номер елементу j , з яким проводиться обмін. По завершенні проходу вниз (вихід блоку 3) зсуваємо ліву межу масиву (блок 6) та виконуємо прохід вгору (блок 7-9).

Покажемо виконання алгоритму для масиву з 7 елементів:

32 9 30 64 14 2 87.

Виконаємо прохід зверху вниз

<i>left</i>	<i>right</i>	<i>k</i>	$left \leq right?$	<i>j</i>	$j > left?$	$a[j - 1] > a[j]?$	Обмен
2	7	7	«Да»	7	«Да»	$2 > 87$ «Нет»	
		6		6	«Да»	$14 > 2$ «Да»	$x = 14; a[5] = 2; a[6] = 14$
		5		5	«Да»	$64 > 2$ «Да»	$x = 64; a[4] = 2; a[5] = 64$
		4		4	«Да»	$30 > 2$ «Да»	$x = 30; a[3] = 2; a[4] = 30$
		3		3	«Да»	$9 > 2$ «Да»	$x = 9; a[2] = 2; a[3] = 9$
		2		2	«Да»	$32 > 2$ «Да»	$x = 32; a[1] = 2; a[2] = 32$
				1	«Нет»		

Масив після цього проходу має вид 2 32 9 30 64 14 87.

Міняємо напрямок руху та виконуємо прохід знизу вгору.

<i>left</i>	<i>right</i>	<i>k</i>	<i>j</i>	$j \leq right$	$a[j - 1] > a[j]?$	
3	7	3	3	«Да»	$32 > 9?$ «Да»	$x = 32; a[2] = 9; a[3] = 32$
		4	4	«Да»	$32 > 30?$ «Да»	$x = 32; a[3] = 30; a[4] = 32$
			5	«Да»	$32 > 64?$ «Нет»	
		6	6	«Да»	$64 > 14?$ «Да»	$x = 64; a[5] = 14; a[6] = 64$
			7	«Да»	$64 > 87?$ «Нет»	
			8	«Нет»		

Тепер масив має вид 2 9 30 32 14 64 87.

Знову міняємо напрям руху. Зміна напрямку руху виконується до тих пір, поки виконується умова, записана в блоці 2.

Аналіз показує [1], що сортування обміном і її невеликі поліпшення гірше, ніж сортування включеннями і вибором. Алгоритм шейкер-сортування рекомендується використовувати в тих випадках, коли відомо, що масив «майже впорядкований».

У 1962 р Чарльз Хоар запропонував метод сортування поділом. Цей метод є розвитком методу простого обміну і настільки ефективний, що його стали називати методом швидкого сортування - QuickSort.

На підставі аналізу, що приводиться в [1], можна зробити наступні висновки про ефективність розглянутих алгоритмів сортування.

1. Перевага сортування бінарними включеннями в порівнянні з сортуванням простими включеннями невелика.

2. Сортування методом «бульбашки» є найгіршою серед методів порівняння. Її поліпшена версія - шейкер-сортування є гіршим, ніж сортування простими включеннями і простим вибором.

3. Сортування простим вибором є кращим з простих методів.

1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. СПб.: Питер, 2017. 288 с.
2. Конова Е. А., Поллак Г. А. Алгоритмы и программы. Язык C++: Учебное пособие. СПб.: Издательство «Лань», 2017. 384 с.: ил.