**Київський національний університет імені Тараса Шевченка**

**Факультет комп’ютерних наук та кібернетики**

**Алгоритми та складність**

**Завдання № 2**

**Звіт**

**Виконав:**

студент групи К-29

Грищенко Юрій Анатолійович

**Київ-2019**

**Умова задачі**

Нехай є *n* болтів різного розміру та *n* відповідних гайок. Припустимо, можна порівнювати, чи підходять гайка і болт одне до одного, або гайка більша (чи менша). Порівняти між собою дві гайки чи два болти неможливо. Розробіть і реалізуйте алгоритм розбивки всіх гайок і болтів на відповідні пари за час Θ(n log n).

**Опис алгоритму:**

Суть алгоритму полягає у тому, що ми фіксуємо один елемент з одного масиву (наприклад, перший болт), і порівнюємо з ним кожну гайку. Одна з цих гайок буде для нього підходити, також матимемо деякі менші гайки та більші гайки. Таким чином отримаємо 2 підмасиви — один буде містити менші гайки та ще один буде містити більші.

Початкові масиви:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Болти | 4 | 5 | 7 | 3 | 6 | 1 | 2 |
| Гайки | 5 | 3 | 4 | 1 | 6 | 2 | 7 |

Беремо перший болт (розмір — 4), і розбиваємо масив гайок:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Болти | 4 | 5 | 7 | 1 | 6 | 3 | 2 |
| Гайки | 2 | 3 | 1 | 4 | 6 | 7 | 5 |

Таким чином, ми знаходимо пару для нашого болта (гайку розміром 4), а також отримуємо масив менших гайок та більших гайок. Це розбиття відбувається за O(n) кроків:

size\_t **sort**(**int** compare\_bolt, **int**\* nuts, size\_t start, size\_t end)

{

**bool** found\_match = **false**;

size\_t i = start;

**while**(i <= end)

{

//якщо гайка більша за наш болт, відсилаємо її в кінець масиву.

**if**(compare\_bolt < nuts[i])

{

std::swap(nuts[i], nuts[end]);

end--;

}

//якщо гайка менша за наш болт і ми не знайшли йому пару, йдемо далі

**else** **if**(!found\_match && compare\_bolt > nuts[i])

{

i++;

**continue**;

}

//якщо гайка менша за наш болт і ми вже знайшли йому пару, то міняємо місцями гайки так, щоб nuts[i] вказував на знайдену «парну» гайку (а менша гайка знаходилась лівіше)

**else** **if**(found\_match && compare\_bolt > nuts[i])

{

std::swap(nuts[i - 1], nuts[i]);

i++;

}

**else** **if**(!found\_match && compare\_bolt == nuts[i])

{

found\_match = **true**;

i++;

}

}

**return** i - 1;

}

Тепер, коли ми розбили масив гайок, робимо те саме з масивом болтів:

Беремо відповідну гайку (розміром 4) і відносно неї розбиваємо масив гайок:

Початкові масиви:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Болти | 4 | 5 | 7 | 3 | 6 | 1 | 2 |
| Гайки | 2 | 3 | 1 | 4 | 6 | 7 | 5 |

Розбили масив болтів:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Болти | 2 | 1 | 3 | 4 | 6 | 7 | 5 |
| Гайки | 2 | 3 | 1 | 4 | 6 | 7 | 5 |

Бачимо, що маємо чітко розбиті підмасиви болтів та гайок, у них однаковий розмір, і вони містять однакові елементи. Для одного елемента пара вже знайдена, отже підмасиви будуть містити n-1 елементів. Тому рекурсивно застосовуємо цей самий алгоритм для кожного з них, доки не матимемо підмасиви довжиною 1.

В середньому випадку, якщо розбиття масивів відбувається більш-менш посередині (тобто кількість підмасивів буде ділитися на 2 з кожним кроком), це буде нагадувати top-down merge sort. Тобто глибина рекурсії буде приблизно log n, отже складність алгоритму O(nlogn).[1]

В найгіршому випадку, якщо розбиття масивів відбуватиметься не посередині (тобто матимемо один підмасив довжиною n-1 і ще один довжиною 0), тоді доведеться виконувати n кроків (і на кожному сортувати спочатку n-1 елементів, потім n-2, n-3 і т. д.) В такому випадку складність буде n + (n-1) + (n-2) + .. + 2 + 1 = n^2 / 2 = O(n^2)

**Інтерфейс користувача**

Користувач вводить довжину масивів та сам вводить розмір кожної гайки (nut) та кожного болта (bolt). Програма потім виводить знайдені пари та їх позиції у масивах.

Enter array length:

7

Enter bolt sizes.

4

{ 4 }

2

{ 4, 2 }

3

{ 4, 2, 3 }

6

{ 4, 2, 3, 6 }

5

{ 4, 2, 3, 6, 5 }

7

{ 4, 2, 3, 6, 5, 7 }

1

{ 4, 2, 3, 6, 5, 7, 1 }

Enter nut sizes.

1

{ 1 }

4

{ 1, 4 }

3

{ 1, 4, 3 }

7

{ 1, 4, 3, 7 }

6

{ 1, 4, 3, 7, 6 }

2

{ 1, 4, 3, 7, 6, 2 }

5

{ 1, 4, 3, 7, 6, 2, 5 }

Bolt size: 4, nut size: 4, index: 3

Bolt size: 2, nut size: 2, index: 1

Bolt size: 1, nut size: 1, index: 0

Bolt size: 3, nut size: 3, index: 2

Bolt size: 7, nut size: 7, index: 6

Bolt size: 5, nut size: 5, index: 4

Bolt size: 6, nut size: 6, index: 5

Nuts array: { 1, 2, 3, 4, 5, 6, 7 }

Bolts array: { 1, 2, 3, 4, 5, 6, 7 }

**Список використаних джерел**

1. <https://en.wikipedia.org/wiki/Merge_sort>