

# 1 Сортиране

Добре познатата на всички задача за сортиране. Като вход на задачата имаме масив  $arr = [a_1 \dots a_n]$  от цели числа. Като изход искаме масивът да е сортиран, тоест  $a_1 \leq a_2 \leq \dots \leq a_n$ . Забележете, че не е задължително да сортираме цели числа. Можем да сортираме всякакви обекти в които имаме дефинирана релацията  $<$ . Също така не е задължително контейнера да е масив - може да е свързан списък или всякакъв контейнер, който поддържа някаква наредба.

Сортирането често е само първата стъпка, много алгоритми работят бързо именно върху сортирани колекции. Добре познатия Ви алгоритъм *binary search* работи за време  $O(\log(n))$  в най - лошия случай за сортиран масив. Като сравнение, ако за вход имаме масив с 1000000 елемента то на алгоритъма ни ще му отнеме ней - много 20 стъпки. Сравнете производителността му с алгоритъма *линейно търсене*.

Това е само един от многото примери защо сортирането е една от най - основните задачи за всеки програмист.

# 2 Стабилност

Едно от важните свойства на сортиращите алгоритми е стабилността. В разглежданите примери сортираме само цели числа, но това далеч не са единствените обекти които можем да сортираме. Често ни се налага да сортираме по - големи записи. Пример за по - голям запис би бил резултатите от произволен изпит. В тези резултати е записана оценката на студента, неговото име, факултетен номер и тн. За различните полета по които можем да сортираме можем да си мислим като ключове.

Стабилен сортиращ алгоритъм е сортиращ алгоритъм, който **не променя разположението на елементи с еднакви ключове**.

Примерно, искам да сортирам студентите положили този изпит по оценки. Напълно възможно да има еднакви оценки. Ако преди това студентите са сортирани по азбучен ред, студентите с еднакви оценки отново ще са сортирани по азбучен ред.

9	3	3'	5	6	5'	2	1	3''
1	2	3	3'	3''	5	5'	6	9
1	2	3'	3	3''	5'	5	6	9

В следния пример първия масив е входен. Втория масив представлява резултатът на стабилен сортиращ алгоритъм Третия представлява резултата на нестабилен такъв.

## 2.1 Insertion sort

Нека си представим, че имаме един сортиран масив  $[a_1 \dots a_k]$ . Сега искаме да вмъкнем някакъв елемент в него така, че той да остане сортиран. Нека наречем този елемент  $a_i$ . Започваме от елемента  $a_k$ . Ако той е по-голям от  $a_i$  го преместваме една позиция надясно. Ако е по-малък то  $a_i$  става последен елемент. Аналогично повтаряме тази идея като изместваме всеки елемент по-голям от  $a_i$  с една позиция надясно. След краен брой стъпки масивът би изглеждал по следния начин  $[a_1 \dots a_i \dots a_k]$  където елементите вдясно от  $a_i$  са по-големи а тези в ляво са по-малки. Тази идея напомня на начина по който картоиграч сортира картите си. В ръка с 4 наредени карти взема петата и я поставя на правилното ѝ място, като по-малките са вляво а по-големите вдясно.

Как ще използваме тази идея за да сортираме целия масив? Ами вече имаме алгоритъм за добавяне на един елемент в сортиран масив. Алгоритъма също така запазва масива сортиран след добавянето на новия елемент. Нека разгледаме несортиран масив  $[a_1 \dots a_n]$ . По дефиниция подмасива  $[a_1]$  е сортиран. Сега ще приложим идеята върху сортирания подмасив  $[a_1]$  и елемента  $a_2$ . След като приложихме идеята знаем, че първите два елемента на масива са сортирани. Нека приложим отново тази идея - този път ще вмъкваме третия елемент в подмасива състоящ се от първите два елемента. Правим това вмъкване  $n$  пъти докато не получим сортиран масив.

Примерен код на *Insertion Sort*:

```

void insertionSort(int* v, size_t size) {
    for (size_t i = 1; i < size; i++) {
        int elem = v[i];
        int j = i - 1;

        while (j != -1 && v[j] > elem) {
            v[j + 1] = v[j];
            j--;
        }

        std::swap(v[j + 1], elem);
    }
}

```

Тук *elem* е елементът, който ще вкарваме в сортирания подмасив а *j* е индексът на най - десния елемент в сортирания подмасив.

Каква е сложността на insertion sort? В най - лошия случай сложността на алгоритъма е  $\Theta(n^2)$ . Каква е сложността по памет на Insertion Sort? Insertion sort е стабилен сортиращ алгоритъм. Стабилността му се дължи на строгото неравенство  $>$ .

## 2.2 Selection sort

Нека разгледаме примерен код на сортирания алгоритъм *Selection sort*:

```

1 void selectionSort(int* arr, size_t size) {
2     for (size_t i = 0; i < size - 1; i++) {
3         size_t minElementIndex = i;
4
5         for (size_t j = i + 1; j < size; j++)
6             if (arr[j] < arr[minElementIndex])
7                 minElementIndex = j;
8
9         if (i != minElementIndex)
10            std::swap(arr[i], arr[minElementIndex]);
11     }
12 }

```

На ред 3 създаваме нова променлива наречена *minElementIndex*. Както името подсказва, тя ще пази в себе си индекса на най - малкия елемент. Добър въпрос е обаче от къде вземаме най - малкия елемент. В началото променливата  $i = 0$ , тоест вътрешния цикъл ще обиколи подмасива  $[1 \dots n - 1]$  и ако има по - малък елемент ще запише индекса му в променливата. След първата итерация на ред 9 имаме в *minElementIndex* е записан индекса на най - малкия елемент в интервала  $[0 \dots n - 1]$ . Естествено, в сортиран масив най - малкия елемент стои в началото. Сега просто пращаме най - малкия елемент в началото. Повтаряме това действие  $n - 1$  пъти. След втората итерация на ред 9 в *minElementIndex* е записан най - малкия елемент в интервала  $[1 \dots n - 1]$ . Неговото място е на индекс 1, та точно там го пращаме с кода на ред 9-10. Аналогично, за  $i = k$  на ред 9 в *minElementIndex* е записан най - малкия елемент в интервала  $[k \dots n - 1]$ .

Сложността по време на *Selection sort* е  $O(n^2)$ . Каква е сложността по памет на *Selection sort*?

Струва си да отбележим, че това не е стабилен сортирац алгоритъм. За да го докажем ще дадем следния контрапример:

2 2' 1

Начален масив

1 2' 2

Масивът след първа итерация

1 2' 2

Изходен масив

Със зелено означаваме минималните елементи. Забелязваме, че алгоритъма не запазва наредбата на елементите с еднакви ключове, следователно той не е стабилен. Изход на стабилен сортирац алгоритъм би бил  $[1, 2, 2']$ . Selection sort е сортиращия алгоритъм който прави най - малко на брой swap - ове. Имаме  $n - 1$  размени на елементи в най - лошия случай. Swap е бърза, но не безплатна операция.

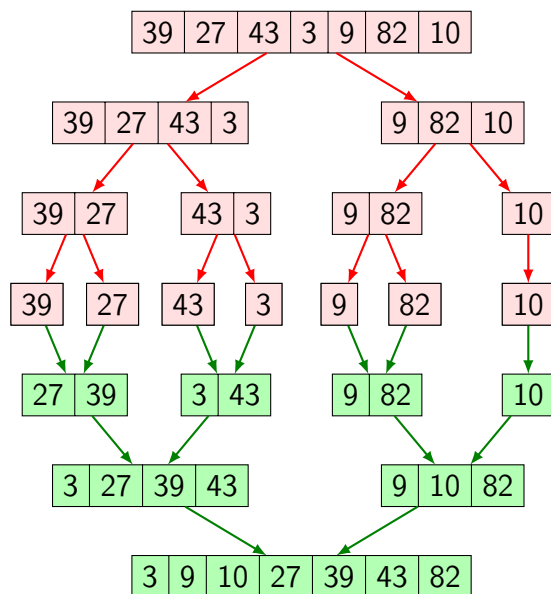
## 2.3 Merge sort

Това ще е първия рекурсивен сортирац алгоритъм който ще разгледаме. Първо, нека се запознаем с функцията *merge*. Тя приема два **сортирани** масива и като резултат създава сортиран масив. Важно е да се отбележи, че масивите са сортирани, ако не са то функцията е безполезна.

Нека имаме входен несортиран масив  $[a_1 \dots a_n]$ . Сега, ако намеря някакъв начин да сортирам подмасивите  $[a_1 \dots a_{\frac{n}{2}}]$  и  $[a_{\frac{n}{2}+1} \dots n]$  мога да извикам функцията *merge* и имам сортиран масив. Но как да ги сортирам?

Ами спомняме си, че пишем сортиращ алгоритъм. Извиквам *mergeSort* **рекурсивно** върху лявата и дясната половина на масива и след като ги сортира викаме функцията *merge*. *mergeSort* ще продължи да разбива масива на две, после левия и десния подмасив на още две и т.н. Масивът е краен, та все някъде трябва да спрем. Спомняме си, че масив от един елемент е тривиално сортиран, това звучи като добро място на което да спрем.

Преди да продължим с обяснението, разгледайте следната диаграма:



Оцветените в зелено клетки са подмасиви на които вече е бил извикан *merge*! Виждате как *mergeSort* разделя масива на части докато не стигне до подмасиви с дължина едно (които играят роля на листа в дървото на рекурсията) след това викаме *merge* на листата, а после на сортираните масиви възпроизведени от листата и т.н.

Нека сега представим примерен код на *mergeSort*

```

1  // arr и arr2 са сортирани масиви
2  void merge(int* arr, size_t lenOne, int* arr2, size_t lenTwo);
3
4  void mergeSort(int* arr, size_t size) {
5      if(size < 2)
6          return;
7
8      size_t mid = size / 2;
9      mergeSort(arr, mid);
10     mergeSort(arr + mid, size - mid);
11     merge(arr, mid, arr + mid, size - mid);
12 }

```

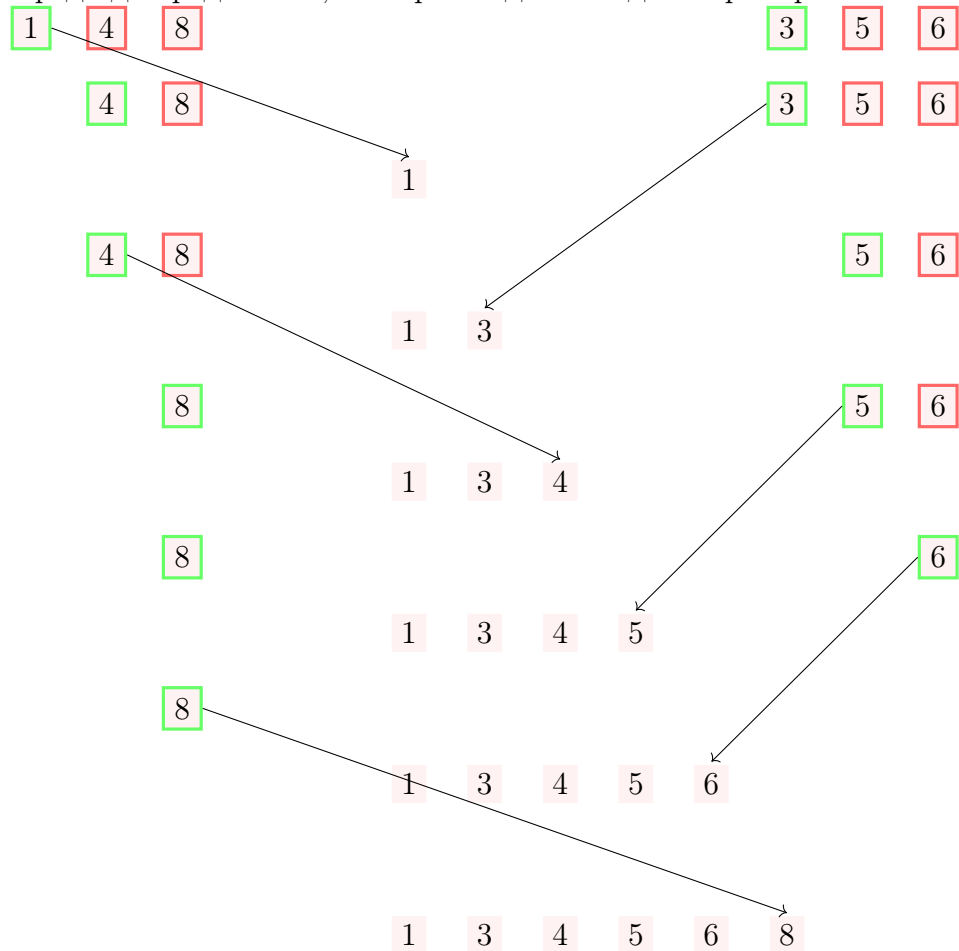
Този код може да изглежда леко объркващ в началото та нека подробно разгледаме какво се случва. На ред 2 просто дефинираме функцията *merge* - с нея ще се занимаваме малко по - късно. На ред 5 правим проверка дали трябва да разбиваме масива още или сме стигнали до дъното на рекурсията. На ред 8 вземаме средата на масива както се разбрахме. На ред 9 и 10 използваме познатата Ви от курса по УП указателна аритметика. Накратко - указателя сочи към **последователна памет**. Тоест вземам лявата половина на масива като пускам указател към началото му, но казвам на *mergeSort*, че дължината му вече е наполовина. Как да взема дясната половина обаче? Ами пускам указател към средния елемент и отново казвам колко е дължината. Как да получа указател към средния елемент? Помним, че ако имаме масив, то името му е указател към първия му елемент. Синтаксисът  $arr[k]$  е синтактична захар за  $*(arr + k)$  където  $arr + k$  връща указател към  $k$ -тия последователен елемент а оператора  $*$  взема стойността му. Тоест когато на ред 10 пишем  $arr + mid$  не правим нищо повече от да си вземем точно указателя от който имаме нужда!

Сега просто извикваме функцията *merge* която обединява двата сортирани масива.

Знаем какво прави *merge* - сега въпросът е как да го реализираме. Оказва се, че най - добрия начин за реализация на *merge* е с допълнителна памет. Нека  $n_1$  и  $n_2$  са дължините на входните масиви. Заделяме нов масив с дължина  $n_1 + n_2$ . Вземаме два индекса които в началото сочат към първите елементи на двата входни масива. Кой би бил най - малкия елемент в новия масив? Ами понеже масивите са сортирани то това ще

е или най - левия елемент на първия масив или най - левия елемент на втория масив. Сравняваме ги и по - малкия записваме на първа позиция. Увеличаваме индекса на масива в който е по - малкия елемент с единица. Сега аналогично итерираме през масивите докато и двата индекса не стигнат края.

Преди да продължим, нека разгледаме следния пример:



Зелените елементи са тези към които сочи съответния индекс. Сивите са вече позиционирани в масива. Виждаме как със сложност  $\Theta(n_1 + n_2)$  по време и памет създадохме нов сортиран масив.

Нека да разгледаме как би изглеждала *merge* на код:

```

1 void merge(int* arr, size_t lenOne, int* arr2, size_t lenTwo) {
2     int* resultArray = new int[lenOne + lenTwo];
3
4     int pntOne = 0;
5     int pntTwo = 0;
6     int resArrayIndex = 0;
7
8     while (pntOne < lenOne && pntTwo < lenTwo) {
9         if (arr[pntOne] <= arr2[pntTwo])
10             resultArray[resArrayIndex++] = arr[pntOne++];
11         else
12             resultArray[resArrayIndex++] = arr2[pntTwo++];
13     }
14
15     while (pntOne < lenOne)
16         resultArray[resArrayIndex++] = arr[pntOne++];
17
18     while (pntTwo < lenTwo)
19         resultArray[resArrayIndex++] = arr2[pntTwo++];
20
21     for (int i = 0; i < lenOne + lenTwo; i++)
22         arr[i] = resultArray[i];
23
24     delete[] resultArray;
25 }

```

While на ред 8 копира елементи в новия масив докато първия или втория входен масив не стигнат края си. Понеже може първия да е по - дълъг от втория или обратното, while циклите на ред 15 и 18 се грижат всеки елемент да отиде на мястото си. След това for цикъла на ред 21 връща сортираните елементи обратно в arr. Понеже паметта е последователна, нямаме проблеми, те ще бъдат на правилните места. И разбира се, изтриваме паметта!

Сложността на рекурсивни алгоритми се определя чрез рекурентни уравнения, но това ще го оставим на друг курс. Сега просто ще кажем, че сложността по време на *mergeSort* е  $\Theta(n \log(n))$ . Сложността по памет е  $O(n)$  понеже никога не заделяме масив по - дълъг от дължината на входния. *mergeSort* е стабилен сортиращ алгоритъм.



## 2.4 Quick sort

Това е втория рекурсивен сортиращ алгоритъм, който ще разгледаме. Идеята е да изберем някакъв елемент (наричан *pivot*) и да пренаредим масива така, че лявата част на масива да се състои от елементите по - малки от *pivot* а вдясно от тях да бъдат елементите по - големи от *pivot*. Каква точно е наредбата в лявата и дясната част **няма значение**. След това алгоритъма бива пуснат рекурсивно върху всяка от двете части.

Функцията, която пренарежда масива по този начин, се нарича ***partition***. Тя приема масив. След изпълнението ѝ масивът ни е с желаната наредба, а функцията ни връща позицията, която разделя по - малките елементи от по - големите. В случая като *pivot* ще избираме средния елемент. Със същия успех бихме могли да изберем последния елемент или първия.

След като имаме позицията и масив с желаната наредба, пускаме сортиращия алгоритъм рекурсивно върху двете части. Дъното на рекурсията е същото като на *Merge sort* - спираме когато масивът има 1 елемент. Кодът би изглеждал по следния начин:

```

1  size_t partition(int* arr, size_t size) {
2      int pivot = arr[size / 2];
3
4      int i = 0;
5      int j = size - 1;
6
7      while (true) {
8          while (arr[i] < pivot)
9              i++;
10         while (arr[j] > pivot)
11             j--;
12
13         if (arr[i] == arr[j])
14             i++;
15
16         if (i < j)
17             std::swap(arr[i], arr[j]);
18         else
19             return j;
20     }
21 }
22
23 void quickSort(int* arr, size_t size) {
24     if (size < 2)
25         return;
26     size_t pivotIndex = partition(arr, size);
27
28     quickSort(arr, pivotIndex);
29     quickSort(arr + pivotIndex + 1, size - pivotIndex - 1);
30 }

```

Нека разгледаме този код. Ред 26 подрежда масива и връща индекса на *pivot* според който после рекурсивно викаме алгоритъма върху лявата и дясната част на масива. Сега остава да разберем как работи *partition*. На ред 2 избираме *pivot* елемента. Избираме си два индекса - единия на първия елемент а другия на последния. Тези индекси вървят един срещу друг докато не открият елементи от масива, които са неправилно разположени т.е.  $arr[i] \geq pivot$  и  $arr[j] \leq pivot$ . Когато намерим тези

елементи, просто ги разменяме. Мястото на левия е в дясната половина и обратното. Понеже е възможно да работим с еднакви числа, кодът на ред 13-14 предотвратява зацикляне. Проверете какво ще стане ако премахнете този код и опитате да сортирате масив  $[1, 10, 1]$ .

Сложността на *Quick sort* в най - лошия случай е  $\Theta(n^2)$ . Има причина обаче този алгоритъм да се нарича *Quick sort*. В общия случай той е по - бърз от други алгоритми за сортиране, със сложност в най - лошия случай  $\Theta(n \log(n))$ . Квадратичната сложност се достига когато всеки път за *pivot* избираме най - малкия или най - големия елемент на масива. Вероятността това да се случи намалява драстично с нарастване на входа. Средната сложност на *Quick sort* е  $\Theta(n \log(n))$ . Въпреки привидно лошата си сложност в най - лошия случай, алгоритъмът почти винаги работи със сложност  $\Theta(n \log(n))$ .

Каква е сложността по памет на алгоритъмът? Алгоритъмът е нестабилен, понеже *partition* не запазва наредбата на елементите с еднакви ключове. Споменахме по - рано, че наредбата не ни интересува, искаме единствено лявата част да е по - малка от *pivot* а дясната по - голяма.

### 3 Долна граница за сортиращи алгоритми използващи директни сравнения

Разглежданите досега сортиращи алгоритми са базирани на директни сравнения. Неформално, директно сравнение е за два елемента  $a_i, a_j$  да си отговорим на въпроса  $a_i < a_j$ . Ако отговорът е да правим нещо ако отговорът е не правим нещо друго и така.

За сортиращи алгоритми използващи директни сравнения имаме долна граница  $\Omega(n \log(n))$ . Това означава, че е невъзможно да се напише алгоритъм, който използва директни сравнения и сортира масива за по - малко от  $n \log(n)$  време.

### 4 Сортиране в линейно време - Counting sort

Нека сега се концентрираме върху следната задача: като вход имаме някакъв масив  $Arr[0 \dots n]$  като за всеки елемент  $x$  от масива знаем, че  $x \in [0 \dots k]$  където  $k$  е някакво естествено число. Примерно, масивът  $[1, 4, 3, 4, 6, 10, 12]$  изпълнява това свойство за  $k = 12$ . Трябва да сорти-

раме масива във време  $O(n + k)$  където  $n$  е големината на масива. Алгоритъма, който решава тази задача се нарича Counting sort. Той използва два работни масива,  $Count[0 \dots k]$  и  $Result[0 \dots n]$ . Нека разгледаме следния код:

```
1  const int K = 100;
2  void countingSort(std::vector<int>& v) {
3      // Създава вектор с големина K и стойности 0
4      std::vector<int> count(K, 0);
5      std::vector<int> result(v.size());
6
7      // O(n)
8      for (size_t i = 0; i < v.size(); i++)
9          count[v[i]]++;
10     // O(k)
11     for (size_t i = 1; i < 100; i++)
12         count[i] += count[i - 1];
13     // O(n)
14     for (int i = v.size() - 1; i >= 0; i--) {
15         result[count[v[i]] - 1] = v[i];
16         count[v[i]]--;
17     }
18     v = result; // може и v = std::move(result), но все пак това е пример
19 }
```

Цикълът на ред 8 е лесен за разбиране - всеки път когато видя елемент със стойност  $i$  увеличи с единица  $count[i]$ . При приключване на този цикъл ще е вярно следното твърдение:

**$count[i]$  е броя на елементите със стойност  $i$  във входния масив.**

Алгоритъмът се казва counting sort, та е очаквано да ги броим в някакъв момент. Но какво става на ред 12?

Преди да разгледаме цикъла на ред 12 трябва да сме убедени в следното твърдение:

**Нека  $x$  е произволен елемент от входния масив. Индекса на  $x$  е (броя на елементите по - малки от него) + (Броя на елементите равни на него) - 1.** Примерно, за масив  $[1, 2, 2, 2, 3, 3, 3]$  правилното мясно на най - дясната тройка е  $4 + 3 - 1 = 6$  (елементите по - малки от него са 4 на брой, тези които са равни са три и вадим 1 заради индексацията).

$count[0]$  пази броя на нулите. Ами от нула няма по - малък елемент според нашето предположение, тоест  $count[0]$  пази в себе си сбора на всички елементи по - малки от 0 (които са 0) и всички елементи равни на нула (което го преброихме в горния цикъл).

Също,  $count[1]$  пази броя на единиците. За да намерим тези, по - малки от него трябва да видим колко са нулите, но това го пише в  $count[0]$ . Ако сте съобразителни ще видите, че при първото влизане в for цикъла на ред 12 при  $i = 1$  имаме  $count[1] + = count[0]$ . Сега в  $count[1]$  са записани елементите по - малки или равни на 1. Правим същото с две, три и така до  $k$ .

След като свършихме с тази магия, можем да твърдим следното:

**Нека  $j$  е произволен индекс от  $[0 \dots k]$ .  $count[j] - 1$  е стойността на най - десния индекс в резултатния масив за елемента  $j$ .**

Сега тръгваме отдясно наляво. Щом за всеки елемент имаме записан коректния най - десен индекс, то в  $result$ , на този коректен индекс записваме елемента. След това намаляме индекса с единица - щом това е коректния индекс за най - десния, то ако има втори той ще стои от негово ляво. Ако няма втори няма какво да се притесняваме, тази стойност няма да бъде достъпена никога повече.

**Важно е защо тръгваме от дясно наляво а не обратното.** Това е за да се запази стабилността на counting sort. Щом пазим най - десния коректен индекс, то ще е грешно да запишем най - левия елемент на него, нали така.

Сложността по време и по памет на разгледания алгоритъм е  $O(n + k)$ .  
Защо?

Сложности	Average –			Memory	Stable
	Best – Case	Case	Worst – case		
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Да
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Не
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Да
Merge	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$	Да
Quick	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(1)$	Не
Count	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(n + k)$	Да

