Понятие за сложност. Нотация O

Различните наследници на интерфейсите Collections и Maps са реализирани с цел по-добро използване на конкурентните ресурси „време“ и „памет“ при решаване на конкретни проблеми. Обикновено се приема, че „по-важният“ ресурс е „време за изпълнение“. Това теоретично не винаги е така, но можем да го приемем като първо практическо приближение.

Използваме по-неточни понятия като „време за изпълнение“, „(времева) ефективност“ за приближение на строго информатичното понятие „(изчислителна) сложност (по време)“ (англ. (computational) complexity). За означаване на това понятие е приета нотацията O (четем „о-голямо“, англ. “big O”).

В течение на програмистката си практика трябва да сме наясно, че някои задачи допускат прилагане на различни алгоритми. Всеки от тях правилно решава проблема, като най-често те се различават именно по ресурсите „памет“ и „време“, които изискват. Ще дадем примери с познати алгоритми за боравене с масиви от елементи.

Ако в задачата е известно, например, че елементите, с които ще боравим, са различни цели числа в интервала от 0 до 1000, можем да използваме масив ***b*** от 1001 променливи от логически тип, всяка от които да получи стойност true, ако елементът с тази стойност принадлежи на конкретните за задачата данни, или false в противен случай. Ако, както е стандартно в Java, всички елементи в този масив имат стойност false в началото, то добавянето на нов елемент със стойност ***x*** към набора става „веднага“ чрез присвояването ***b***[***x***]=true; Проверката дали даден елемент присъства в набора също е „непосредствена“ – съответна на логическия израз ***b***[***x***]==true (или просто ***b***[***x***], тъй като самото то е логически израз). Такива алгоритми наричаме „константни“ по време и това е прието да се означава с нотацията O(1). Какъв ресурс отнема подреждането на такава структура? Ами „нула време“ (елементите са си естествено подредени нарастващо) и памет – 1001 пъти по големината на елемент от класа Boolean. Тъй като информацията за принадлежност на всеки елемент е един бит (или го има, или го няма), ефективността по памет може да се подобри, ако се използва точно един бит (с малко повече време за програмиране, но без съществена промяна на времето за добавяне и достъп). Обикновено в такава задача се счита, че това е „излишен труд“, но с нарастването на броя на възможните елементи (не до 1000, а до 1000000000, например) икономията на ресурса памет може да се окаже важно. Ако може да има повтарящи се стойности, този алгоритъм може да се модифицира без съществени загуби на ресурс време, като вместо логически тип се използва целочислен – на всеки допустим елемент съответства „брояч“ – колко пъти се среща.

Ако елементите са по-малко на брой цели числа, но в много голям обхват (напр. не повече от 1000, но от клас Long) или пък от друг, нецелочислен клас, такъв „естествен“ алгоритъм става невъзможен. В такива случаи използваме други структури: масиви, списъци, множества, таблици… Някои от операциите с такива структури изискват „търсене“ – например, ако масив от числа вече е подреден, търсенето в него може да става двоично: проверява се елементът „в средата“ и той или е там, или в зависимост от търсеното „едната половина“ от масива отпада от разглеждане, а с другата процедираме по същия начин. Така за тест за наличие в масив от ***n*** елемента се изискват проверки, чийто брой в най-лошия случай е от порядъка $\log_2(\boldsymbol{n})$ (за масив от 1000 елемента – не повече от 10 проверки). Алгоритми от този характер имат „логаритмична сложност“ по време и това е прието да се означава с O(log(***n***)).

Ако масивът не е подреден, тестът за наличие или липса на елемент в него ще изисква, в най-лошия случай, пълното му „обхождане“, т.е. ***n*** проверки.

Подреждането на масив от ***n*** елемента с алгоритъма „пряк избор“ изисква пък $\frac{n(n-1)}{2}$ проверки.

Алгоритми, броят на операциите в които е полином от тяхната големина, се наричат „полиномиални“ и е прието да се означават с O($\boldsymbol{n^p}$), където ***p*** е степента на полинома. Така пълното обхождане е „линеен алгоритъм“ със сложност O(***n***) (***p***=1,както е прието в математиката, се подразбира като степен и може да не се записва), подреждането чрез пряк избор е „квадратичен алгоритъм“ със сложност O($\boldsymbol{n^2}$), срещат се „кубични“ и други полиномиални алгоритми.

Познаваме и алгоритми с по-особена зависимост от броя елементи. „Бързото сортиране“, например, е по-добро от „метода на мехурчето“ (който, всъщност, пак е квадратичен, като „прекия избор“), тъй като чрез „разделянето“ на масива на две добре наредени относно ключовия елемент части (макар и вътрешно неподредени), единият множител на $\boldsymbol{n^2}$=***n*****n*** се замества с log(***n***). За такава сложност пишем O(***n.***log(***n***)).

Други познати алгоритми изискват „пълно изчерпване“ (backtrack), например – обхождане на лабиринт. Такива алгоритми имат „експоненциална сложност“ O($\boldsymbol{a^n}$), където ***a*** е константа (често ***a***=2). Работата с всички подмножества на дадено множество от ***n*** елемента (които са $2^{\boldsymbol{n}}$ на брой) е пример за сложност O($2^{\boldsymbol{n}}$). Работата пък с всички подредби на елементите от това множество (които са ***n***!=1.2.3…..***n*** на брой) е пример за „факториелна сложност“ O(***n***!).