УДК 519.712.3 **Информационные технологии**

Базеева Наталья Алексеевна, *преподаватель факультета довузовской подготовки и среднего профессионального образования, ФГБОУ ВО «Мордовский государственный университет им. Н.П. Огарёва»*

Овтин Руслан Александрович, *студент факультета довузовской подготовки и среднего профессионального образования, ФГБОУ ВО «Мордовский государственный университет им. Н.П. Огарёва»*

НОТАЦИЯ BIG O И СЛОЖНОСТЬ АЛГОРИТМОВ

Аннотация: в данной статье рассматривается нотация Big O в целях изучения сложности алгоритмов в программировании, исследуются «worst case» для некоторых алгоритмов и случаи применения.

Ключевые слова: алгоритм, Big O, сложность алгоритма, «worst case», массив, операция.

Abstract: in this article discusses the Big O notation in order to study the complexity of algorithms in programming, explores the "worst case" for some algorithms and use cases.

Key words: algorithm, Big O, complexity of algorithms, «worst case», array, operation.

Нотация Big O служит для обозначения верхней границы сложности алгоритма. Это идеальный инструмент для нахождения «худших случаев» работы алгоритмов.

Big O призвана показать, как сильно увеличивается количество операций при увеличении размера данных.

Возьмем, к примеру, любой массив, нашей целью будет получить значение второго элемента массива, т.е. нужно будет обратиться ко второму

члену. Какая сложность у этого алгоритма? Он очень прост, нам даже не важен размер массива, т.к. в любом случае нам всегда понадобится лишь одна операция. В таком случае говорят, что сложность алгоритма O(1) или, что алгоритм выполняется за постоянное время. Такие алгоритмы самые эффективные.

Код на C#:

```
int[] array = new int[10] { 10, 9, 1, 2, 6, 3, 6, 7, 5, 2 };
Console.WriteLine(array[1]);
```

Для Big O не важно фактическое число шагов, важно лишь то, что алгоритм выполняется за константное время. Поэтому если алгоритм имеет 5 операций, о нем не говорят, как O(5), а все также считают за O(1).

Есть случай по сложнее. Допустим, что нам нужно вычислить сумму всех элементов массива. Таким образом нам потребуется одна операция на каждый член массива, а это значит, что в нотации Big O сложность будет записана как: O(n). Такие алгоритмы называют линейными или линейно масштабируемыми. Если в каком-то случае фактическая сложность алгоритма: O(n+5), то в Big O нотации это O(n) [2].

Код на C#:

```
int[] array = new int[10] { 10, 9, 1, 2, 6, 3, 6, 7, 5, 2 };
int sum = 0;
foreach(int num in array) // для простоты понимания использован цикл foreach
{
   sum += num;
}
Console.WriteLine(sum);
```

Давайте теперь рассмотрим случай, когда нам необходимо пройтись по всем элементам двумерного массива и, например вывести их на экран. Для простоты представим, что количество строк и количество значений в строке равны. Таким образом нам понадобится уже два цикла, причем один из циклов вложен в другой (в первом цикле мы проходимся по строкам, а во втором по каждому значению в ней). Так сложность нашего алгоритма равна: O(n*n) или $O(n^2)$.

Код на C#:

```
int[,] array = new int[,] { {1, 5, 6}, {10 , 14, 2}, { 3, 12, 19},
                            {19, 55, 766}, {101 , 114, 22}, { 7, 52, 29},
                            {5, 3, 67}, {70 , 54, 2}, { 43, 132, 119}};
for(int i = 0; i < array.GetLength(0); i++)
{
   for(int j = 0; j <array.GetLength(1); j++)
   {
      Console.Write(array[i, j] + " ");
   }
   Console.WriteLine();
}
```

Рассмотрим некоторые известные алгоритмы со стороны Big O. Существует алгоритм бинарного поиска, его используют в тех случаях, когда массив отсортирован, а линейные алгоритмы применять не рационально. Суть алгоритм в делении массива на две части и отбрасывания той, в которой требуемое нам значение оказаться не может. В худшем случае мы делаем столько операций на сколько раз мы можем разделить массив на две части, т.е. если в нашем массиве 8 элементов, то 3 раза, а если 16, то 4. Это значит, что сложность алгоритма $O(\log_2(n))$ или просто $O(\log(n))$. Алгоритм возвращает индекс искомого элемента массива.

Код на C#:

```
int[] array = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
static int BinarySearch(int[] array, int searchedValue)
{
   int left = array.GetLowerBound(0);
   int right = array.GetUpperBound(0);
   while (left <= right)
   {
      var middle = (left + right) / 2;

      if (searchedValue == array[middle])
      {
         return middle;
      }
      else if (searchedValue < array[middle])
      {
         right = middle - 1;
      }
      else
      {
```

```
            left = middle + 1;
        }
    }
    return -1;
}
Console.WriteLine(BinarySearch(array, 7)); // На консоль выведется 6
```

Иногда бывают комбинированные алгоритмы, в которых используются сразу несколько алгоритмов попроще. Например, сложность O(nlog(n)) мы получаем в том случае, если на пару с алгоритмом перебора мы используем бинарный поиск. В пример приведен алгоритм, который выводит на консоль элементы индекс, которых совпадает со значением, естественно такое действие можно выполнить гораздо проще, но в данном случае нам важна наглядность способа.

Код на C#:

```
int[] array = new int[] { 2, 2, 3, 12, 5, 6, 72, 8, 9, 10 };
static int BinarySearch(int[] array, int searchedValue)
{
    int left = array.GetLowerBound(0);
    int right = array.GetUpperBound(0);
    while (left <= right)
    {
        var middle = (left + right) / 2;
        if (searchedValue == array[middle])
        {
            return middle;
        }
        else if (searchedValue < array[middle])
        {
            right = middle - 1;
        }
        else
        {
            left = middle + 1;
        }
    }
    return -1;
}

for (int i = 0; i < array.Length; i++)
{
    Console.WriteLine(BinarySearch(array, i)); // Элементы, у которых индексы не совпадают значению,
```

выведут на консоль -1

}

Самыми плохими по количеству операций являются алгоритмы со сложностями O(n!), $O(n^k)$ и $O(2^n)$. Причиной служит то, что они возрастают с невероятной скоростью. Обычно, если наблюдается такая высокая сложность это означает неэффективность подобранного алгоритма.

Важным моментом является то, что нотация Big O рассматривает алгоритмы со стороны «worst case», т.е. со стороны худшего случая для выбранного алгоритма. Помимо него существуют «average case» и «best case», средний и лучший случаи соответственно. Например, для сортировки пузырьком лучшим случаем является O(n), когда массив уже отсортирован. Также существуют случаи, когда сложности для случаев совпадают, например, у сортировки вставками сложности для «worst case», «average case» и «best case» всегда O(log(n)) [3].

В приведенном ниже списке рассматриваются сложности некоторых часто используемых алгоритмов сортировки:

- Пузырьковая сортировка – $O(n^2)$;
- Сортировка вставками – $O(n^2)$;
- Сортировка выбором – $O(n^2)$;
- Быстрая сортировка – O(log(n));
- Сортировка слиянием – O(log(n));
- Пирамидальная сортировка – O(log(n)).

Важно понимать, что Big O всего лишь теоретическая оценка алгоритма, в реальности эффективность кода зависит от огромного множества различных факторов, например производительности компьютера или операционной системы.

Нужно знать, что быстрота кода часто приводит к его усложнению, поэтому не стоит всегда рассматривать алгоритмы со стороны производительности, понятность и простота не менее важные показатели, особенно когда программирование происходит в команде.

Библиографический список:

1. Big O [электронный ресурс] habr.com – режим доступа: https://habr.com/ru/post/444594/.

2. Шпаргалка по Big O Notation: быстрые ответы на вопросы Big O [электронный ресурс] bestprogrammer.ru – режим доступа: https://bestprogrammer.ru/izuchenie/shpargalka-po-big-o-notation-bystrye-otvety-na-voprosy-big-o.

3. What is Big O Notation Explained: Space and Time Complexity [электронный ресурс] www.freecodecamp.org – режим доступа: https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/.