

**“AZƏRBAYCAN HAVA YOLLARI” CJSC NATIONAL AVIATION ACADEMY**

**Individual Work № 2:**

**Topic: Space Complexity**

**Subject: Obyektyönümlü proqramlaşdırma**

**Teacher: Mamed Shahmaliyev**

**Group: 1459i Student: Mehrali Babayev**

**Date: 11.10.2021 Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_**

**­ Baku 2021**

**What is Space Complexity ?**

The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity.

*Auxiliary Space* is the extra space or temporary space used by an algorithm.

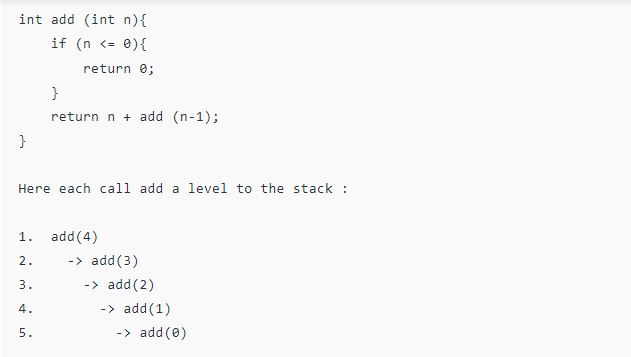
*Space Complexity*of an algorithm is the total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criterion than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort, and Heap Sort use O(1) auxiliary space. The space complexity of all these sorting algorithms is O(n) though.

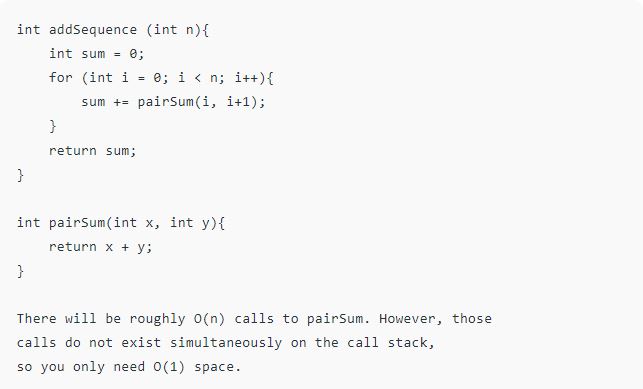
Space complexity is a parallel concept to time complexity. If we need to create an array of size n, this will require O(n) space. If we create a two-dimensional array of size n\*n, this will require O(n2) space.

In recursive calls stack space also counts.

Example :

 However, just because you have n calls total doesn’t mean it takes O(n) space.

Look at the below function :

****

**Note:** It’s necessary to mention that space complexity depends on a variety of things such as the programming language, the compiler, or even the machine running the algorithm.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**1. Introduction**

**Space complexity measures the total amount of memory that an algorithm or operation needs to run according to its input size.**

In this tutorial, we’ll see different ways to quantify space complexity. Moreover, we’ll analyze the total space taken via some examples.

Lastly, we’ll discuss how space and time complexity impact each other.

## 2. Notations

There are different notations we can use to express space complexity measurements. The most widely used is [big-O notation](https://www.baeldung.com/cs/big-o-notation), and that’ll be our main focus. Also, we’ll briefly define other common notations.

### 2.1. Big-O Notation

Big-O notation describes an asymptotic upper bound. It represents the algorithm’s scalability and performance.

Simply put, **it gives the worst-case scenario of an algorithm’s growth rate.**We can say that: “the amount of space this algorithm takes will grow no more quickly than this f(x), but it could grow more slowly.”

Let’s see a few examples of expressing space complexity using big-O notation, starting from slowest space growth (best) to fastest (worst):

1. O(1) – constant complexity – takes the same amount of space regardless of the input size
2. O(log n) – logarithmic complexity – takes space proportional to the log of the input size
3. O(n) – linear complexity – takes space directly proportional to the input size
4. O(n log n) – log-linear/quasilinear complexity – also called “linearithmic”, its space complexity grows proportionally to the input size and a logarithmic factor
5. O(n^2) – square/polynomial complexity – space complexity grows proportionally to the square of the input size

### 2.2. Omega Notation – Ω

Omega notation expresses an asymptotic lower bound.

So, **it gives the best-case scenario of an algorithm’s complexity**, opposite to big-O notation. We can say that: “the amount of space this algorithm takes will grow no more slowly than this fix), but it could grow more quickly.”

Let’s analyze a simple example to illustrate why we prefer big-O notation over omega notation. Using Ω, we could say that the richest person on Earth owns at least $10. It is a true sentence but obviously far from being precise.

### 2.3. Theta Notation – θ

Theta notation represents a function that is within lower and upper bounds.

We can say that: “the algorithm’s space takes at least that (lower bound function) amount of space and no more than that (maximum bound function) amount of space”.

## 3. Analyzing Space Complexity of Algorithms

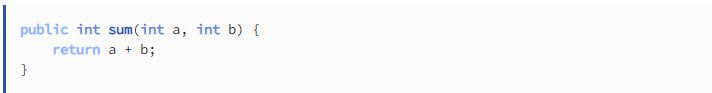
The ability to calculate space complexity is essential in considering an algorithm’s efficiency. In this section, we’ll analyze the space complexity of a few programs of differing difficulty. We’ll measure it using big-O notation.

Above all, it’s necessary to mention that **space complexity depends on a variety of things such as the programming language, the compiler, or even the machine running the algorithm**.

We’ll use Java for our examples, although, each of them could be easily applied to any technology. To apply them to other programming languages, we can simply replace the Java data types with language-specific ones. Then in our calculations, we’d use the sizes that correspond to the language-specific data types.

For example, to port the below Java example JavaScript, we’d replace Java’s *int* with JavaScript’s *number*. And, since Java’s *int*occupies four bytes and JavaScript’s *number* occupies eight bytes, we’d adjust the data sizes used in our analysis accordingly.

To get warmed up, let’s consider a simple Java operation that sums two integers (numbers without a fractional part):



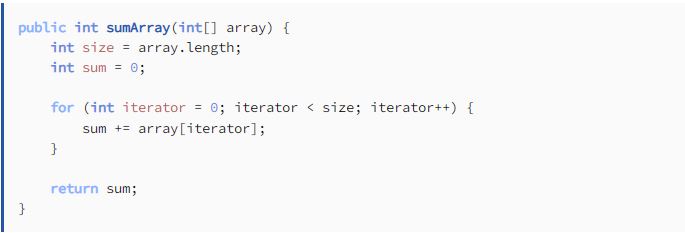
In this particular method, three variables are used and allocated in memory:

* the first *int* argument, *a*
* the second *int* argument,*b*
* and the returned sum result which is also an *int*like *a*and *b*

In Java, a single integer variable occupies four bytes of memory. In this example, we have three integer variables. Therefore, this algorithm always takes 12 bytes of memory to complete (3\*4 bytes).

We can clearly see that the space complexity is constant, so, it can be expressed in big-O notation as *O(1)*.

Next, let’s determine the space complexity of a program that sums all integer elements in an array:



Again, let’s list all variables present in the above code:

* *array* – the function’s only argument – the space taken by the array is equal 4*n* bytes where *n* is the length of the array
* *size* – a 4-byte integer
* *sum* – a 4-byte integer
* *iterator* – a 4-byte integer

The total space needed for this algorithm to complete is 4*n* + 4 + 4 + 4 (bytes). The highest order of *n* in this equation is just *n*. Thus, the space complexity of that operation is *O(n)*.

## 4. Space-Time Complexity Tradeoffs

All efforts made by analyzing time and space complexity lead to the algorithm’s efficiency.

But, when we can say that an algorithm is efficient? The answer seems to be obvious: it should be fast, and it should take the least amount of memory possible.

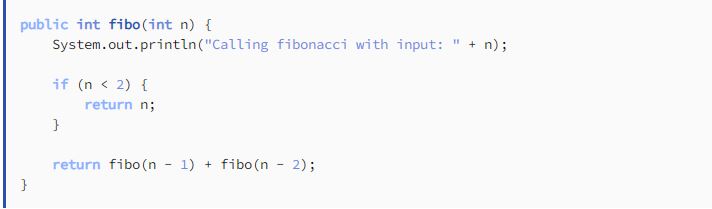
Unfortunately, in algorithmics, space and time are like two separate poles. **Increasing speed will most often lead to increased memory consumption and vice-versa**.

On the one side, we have merge sort, which is extremely fast but requires a lot of memory. On the other side, we have bubble sort, a slow algorithm but one that occupies minimal space. There are also some balanced ones like in-place heap sort. Its speed and space usage are not the best, but they’re acceptable.

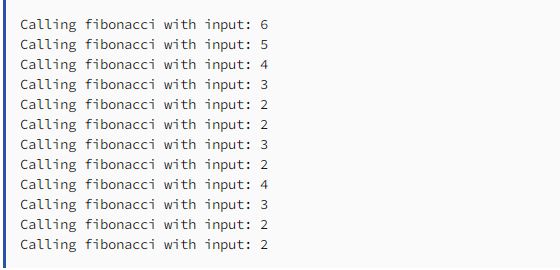
### 4.1. Practical Example

A great example of optimizing the time complexity of the algorithm at the expense of memory is [memoization](https://en.wikipedia.org/wiki/Memoization). It’s a technique used to reduce time complexity of algorithms that frequently call some method with the same input data. Instead of calling the method every time, which might be time-consuming, we store the results, and on each call, we check if there is already a cached result for a given input.

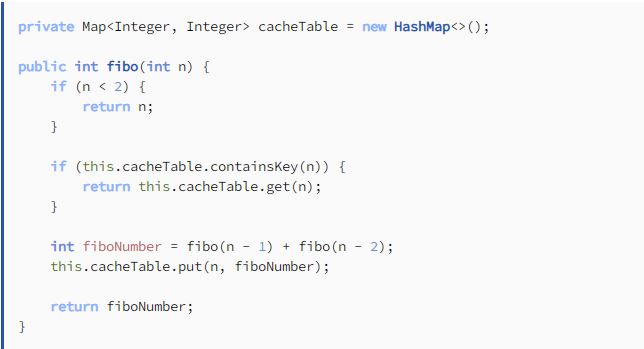
Let’s see that in action. Below, we can see an algorithm that calculates a [Fibonacci number](https://en.wikipedia.org/wiki/Fibonacci_number):



It’s a recursive solution, and its time complexity is O(2^n). After running it with input n = 6, we can see the console output:



As we can see, the method is called with the same input a few times. Let’s optimize it using memoization:



In the above algorithm, we’re calling the method only once for each input number. Thus, we reduced complexity to O(n). At the same time, we increased memory used by the program with the addition of a HashMap that is storing previously-calculated pairs of input n and the corresponding result.