| Array size | Execution run time in microseconds | Function run | | | |
|---|---|---|---|---|---|
| | doublerInsert(.unshift) | doubleAppend(.push) | Times Faster | Faster function | |
| tiny Array (10) | 39.4 | 98.3 | 2.49 | unshift | |
| smallArray (100) | 50 | 105.5 | 2.11 | unshift | |
| medium (1000) | 192.7 | 207.3 | 1.08 | unshift | equal |
| Large (10,000) | 10276 | 145.8 | 70.48 | push | |
| extra large (100,000) | (1s)1201762.8 | (3ms)3236.7 | 371.29 | push | 1 |
| Double extra large (200,000) | (4.9 s)4900000 | (5.9 ms) 5900 | 830.51 | push | 2.2 |

Append function uses the .push method  that adds the new item in the very end of the array.

Insert function uses .unshift that adds element in the very start of array and shifts the rest of the array by +1.

Run Time complexity

The append function starts slower but as the array size increases to thousands of items, the append function becomes faster with respect to insert/. unshift function. Append/push scales at O(n) linearly. The insert function using unshift method scales with runtime complexity of O(n^2). So, Append/ push scales better and should be used if the array size is unknown as array can be larger having millions of items. The unshift will be better used if array size is in 100s. For a1000 array size equal in run time.

Execution run time when an extra-large size array of 100000 items is passed the insert took 1.2 seconds vs the append took 3.2 milliseconds i.e.  unshift was 400 times faster.

On doubling the extra-large size array the function become 2 times faster i.e. quadratic function O(n^2).

Space complexity

The insert function using unshift method is slower than append(push) as it has to assign new memory places for all existing values prior to being able to place the new value at the beginning of the array. If the array size is small, it is alright to use it but as the array becomes larger having thousands or more of values, it becomes taxing for the system to do that much memory allocation.

---------------------xxxx---------------------------

In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

push() has a Constant Time Complexity and so is O(1). All it does is add an element and give it an index that's 1 greater than the index of the last element in the array. So it doesn't matter whether the array has 10 elements or 1000. The number of operations that needs to be performed won't change.

Unshift is slower than push because it also needs to unshift all the elements to the left once the first element is added.

## Adding Or Removing An Element

Let’s start with adding. The most common ways I can think of to add an element to an existing array are with the *Array.push()* method which adds an element at the end of an array and the *Array.unshift()* method which adds an element to the beginning of an array. You may think they work the same way and so should have the same Time Complexity. That’s however not true. The **Array.push()** has a **Constant Time Complexity** and so is **O(1).** All it does is add an element and give it an index that’s 1 greater than the index of the last element in the array. So it doesn’t matter whether the array has 10 elements or 1000. The number of operations that needs to be performed won’t change. The same however cannot be said about *Array.unshift*(). Adding an element at the beginning of an array means the new element will have an index of 0. Which means that the index of every other element must be incremented by 1. So **Array.unshift()** has a **Linear Time Complexity** and is **O(n).** The *Array.pop()* and *Array.shift()* methods which are used to remove an element from the end and beginning of an array respectively, work similarly. **Array.pop()** is **O(1)** while **Array.shift()** is **O(n)**. We can use the **Array.splice()** method to remove an element and/or insert elements at any position in an array. When we use this method, the number of indices that need to be changed depend on which index you splice. But in the worst case scenario which is if you splice at the very start is **O(n).** Technically, the Big O for this is **O(n + m)** where n depends on arr1’s length and m on arr2's.

## Time Complexity of two for loops [duplicate]

```
for(i;i<x;i++){
  for(y;y<x;y++){
    //code
  }
}
```

**is n^2**

but would:

```
for(i;i<x;i++){
  //code
}
for(y;y<x;y++){
  //code
}
```

be **n+n?**

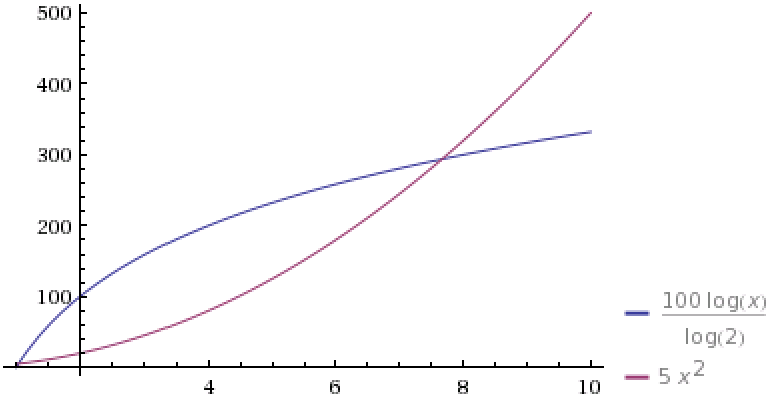
**o(n)**

Since the big-O notation is not about comparing absolute complexity, but only relative one, O(n+n) is in fact the same as O(n). Each time you double x, your code will take twice as long as it did before and that means O(n). Whether your code runs through 2, 4 or 20 loops doesn't matter, since whatever time it takes for 100 elements, it will take twice the time for 200 elements and 100 times the time for 10'000 elements, no matter how much time of that is spent within which loop.

That's why big-O is not saying anything about absolute speed. Whoever assumes that a O(n^2) function f() is always slower than a O(log n) function g(), is wrong. The big-O notation only says that if you keep increasing n, there will be a point where g() is going to overtake f() in speed, however, if n always stays below that point in practice, then f() can always be faster than g() in real program code.

Example 1
Let's assume f(x) takes 5 ms for a single element and g(x) takes 100 ms for a single element, but f(x) is O(n^2), g(x) is O(log2 n). The time graph will look like this:

Plot:



Note: Up to 7 elements, f(x) is faster, even though it is O(n^2).