* **Code Movement** :
  + The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
  + Such a code unnecessarily gets execute again and again with each iteration of
  + the loop. This leads to the wastage of time at run time.
* **Dead Code Elimination:**
  + As the name suggests, it involves eliminating the dead code.
  + The statements of the code which either never executes or are unreachable or their output is never used are eliminated.
* Multiplication with powers of 2 can be replaced by shift right operator which is less expensive than multiplication.
* **Variable propagation**
  + **Example:**
  + Before Optimization After Optimization
  + c = a \* b c = a \* b
  + x = a x = a
  + till till
  + d = a \* b + 4 d = x \* b + 4
* **Unreachable code elimination:**
  + **Example**
    - def collision(bullx, bully, enemyx, enemyy):
    - distance = math.pow(enemyx - bullx, 2) + math.pow(enemyy -bully, 2))
    - if distance < 40:
    - return True
    - else:
    - return False
    - if collision(bullx, bully, enemyx, enemyy):
    - quit()

* **Loop Jamming:**
* Two or more loops are combined in a single loop. It helps in reducing the compile time

# Optimizing Python Code:

* **Use builtin functions and libraries:**

Builtin functions like map() are implemented in C code. So the

Interpreter doesn’t have to execute the loop, this gives a

considerable speedup.   
 The map() function applies a function to every member of iterable

and returns the result. If there are multiple arguments, map()

returns a list consisting of tuples containing the corresponding

items from all iterables.

* **Use keys for sorts:**

In Python, we should use the key argument to the built-in sort

instead, which is a faster way to sort.

# Python program to illustrate

# using keys for sorting

somelist = [1, -3, 6, 11, 5]

somelist.sort()

print (somelist)

s = 'geeks'

s = sorted(s)

print (s)

* **Use**[xrange](https://www.geeksforgeeks.org/range-vs-xrange-python/)**instead of range:**

range() – This returns a list of numbers created using range()

function.

xrange() – This function returns the generator object that can be

used to display numbers only by looping. Only particular range is

displayed on demand and hence called “lazy evaluation”.

* **Use Python multiple assignment to**[swap variables](https://www.geeksforgeeks.org/how-to-swap-two-variables-in-one-line/)**:**

This is elegant and faster in Python

* **Use local variable if possible:**

  Python is faster retrieving a local variable than retrieving a global

variable. That is, avoid the “global” keyword. So if you are going to

access a method often (inside a loop) consider writing it to a

variable.

* **Try to use Efficient Algorithms:**

In most cases using dynamic programming approach instead

of recursion decreases the time complexity from O(2^n) to O(n)

and the space complexity remains the same.

Use of binary search instead of linear search reduces the time

Complexity from O(n) to O(logn).

Note: The array should be sorted

### Apply the Peephole Optimization Technique

<Peephole> is a code optimization technique in Python that is done

at compile time to improve your code performance. With the

Peephole optimization technique, code is optimized behind the

scenes and is done either by pre-calculating constant expressions or

by using membership tests.

### Profile Your Code

By profiling your code, you can identify areas of

improvement in your code for further optimization. There

are two main ways to do this:

#### **1.Use <timeit>**

#### **2. Use <cProfile>.**

# Optimize C++ Code:

## Use aligned memory allocation:

Instead of malloc(), use aligned\_alloc().

aligned\_alloc() allocates aligned memory which helps C++ code to take advantage of cache and perform optimally.

aligned\_alloc(size\_t alignment, size\_t size)

C++

Copy

alignment is usually a higher power of 2. Memory returned by malloc is an alignment of 8.

## **Optimize Memory allocation:**

Use libraries like TCMalloc to optimize the use of malloc. TCMalloc stands for thread caching malloc.

## Always free dynamically allocated memory:

Free dynamically allocated memory as soon as possible within your code. This help avoid out of memory issues and perform optimally.

## Use third-party optimized libraries:

Use third-party optimized libraries such as Eigen, OpenBLAS, FLAME BLIS, MKL if your code requires a function implemented by these libraries.

The most common library is GEMM for optimized matrix multiplication from the above libraries.

## Pass by reference (not value):

When you pass by value, the object is copied and a new memory is allocated. Memory allocation is always an overhead and in this case, the data is copied as well.

## **Avoid casting:**

Always avoid casting as it is one of the most expensive operations. Define with the datatype that is required.

## Use Inline keyword:

Define all functions as inline. This will move all function code to the area where the function is called and the overhead from function call will be eliminated.

## **Use memset, memcpy:**

If you need to initialize an array with the same value, then use memset instead of looping through all values and setting them. memset is an intrinsic so it is exponentially fast.

memset(array, value, size\_of\_array);

C++

Copy

Similarly, if you need to copy an array to another array, then use memcpy.

memcpy(destination\_array, src\_array, size\_of\_src\_array);

## **Use qsort and bsearch:**

To sort any array in C++ even with custom objects, use qsort(). It is an optimal implementation of QuickSort that is supported within C++ as a standard library.

qsort(array, number\_of\_elements, sizeof(int), comparison\_function);

C++

Copy

Similarly, for Binary Search, use bsearch().

bsearch (array, number\_of\_elements, search\_value, sizeof(int)

, comparison\_function);

C++

Copy

These are usually, faster than manual implementation.

## **Minimize Read and Write I/O:**

Use buffer for read and write. For output, prepare the output as a string and print it once instead of multiple print statements.

Do not use std::endl in print statements. Instead use new line character "\n".

## **Use Initialization over Assignment**

When an object is initialized, then copy constructor is invoked.

datatype object = new datatype(value);

C++

Copy

If an object is declared, then the default constructor is invoked and later, when the object is assigned a value, the copy constructor is invoked.

datatype object;

...

object = new datatype(value);

C++

Copy

So, if an object is initialized directly, then the cost of involing default constructor can be saved.

## Use reference instead of pointer

One should use reference instead of pointer as compiler find it difficult to optimize code using pointers as it is not efficient to determine if two pointers refer to the same location.

void Ptr(const int\* p) { x += \*p; }

void Ref(const int& p) { x += p; }

C++

Copy

Additionally, reference does not require dereference operator and does not need additional case of handling NULL. So, it is better in terms of code cleanup as well.

# JavaScript Optimization:

## **Order in which elements are loaded:**

First, it’s important that all elements in the <head> section are pre-loaded, before the visitor sees anything in-browser, then all subsequent elements are ordered to load in a logical way. Any JavaScript inside the <head> section can slow down a page’s rendering. Here’s a look at the difference between an optimized and an unoptimized page load:</head></head>

When an unoptimized page is loading, chances are a user will see a “white screen” before the full page is then loaded. An optimized page load (render actually) happens in a more step-by-step way, allowing a user to see some content gradually until the page loads fully.

## **Minify JavaScript code for smaller file sizes:**

Minifying code is different from obfuscating code, but both are methods of transforming JavaScript—to be more difficult to read, or to make it smaller. Minification accomplishes the latter, and can shrink file sizes to decrease page load times.

Line breaks, additional spaces, comments etc.—all of this increases the size of a JavaScript file and affects the speed of page load. Compressing the code solves this issue well. Machines are not sensitive to the visual style of code like a human would be. Computers can read and launch minified code, even if all of your Javascript fits in just one string.

## **Exclude unused components of .js libraries:**

Most developers use libraries like jQuery UI or jQuery Mobile as is. This means that the code includes all possible components of each library, when you may only need two or three. A similar situation occurs with other JavaScript libraries as well. If you have the ability to manage what components will be included in your package of library, definitely do it. Your website will load much faster, and your visitors will get a better experience.

## **Use the HTTP/2 protocol**

This second, encrypted version of the main Internet protocol can provide you with a lot of cool features, including the asynchronous download of external files, most notably JavaScript. While HTTP requires deep learning and an advanced knowledge of JavaScript theory, [HTTP/2](https://www.upwork.com/resources/what-is-http2) can make JavaScript load faster

## **Where you can, use CSS3 effects in place of JavaScript:**

Older versions of CSS like 1.0 or 2.0 were not as powerful on their own and required a bit more JavaScript to create more advanced styling effects. But CSS 3.0 is a very capable language on its own, with lots of added functionality that requires less JavaScript. Another benefit is that CSS can be pre-compiled so CPU usage for CSS is lower than JavaScript.

## **Memory leaks**

A memory leak refers to a situation where an application finishes using memory but does not return it back to the underlying OS to be used by another application or process. Every time you create an object or variable in JavaScript, memory is consumed. JavaScript memory leaks occur when you are technically finished with an object or variable but the JS runtime still thinks you need it.This can cause a drag on system performance as resources that should otherwise be freed up for other processes and applications are no longer available.

The best way to avoid JavaScript memory leaks is to properly manage your scope. Here are some common examples of how memory leaks occur in JavaScript:

### Accidental global variables

Globally scoped variables are accessible to all scripts and functions in a JavaScript document. For this reason global variables are not automatically cleaned up by JavaScript’s garbage collector. It’s important to use global variables sparingly and to remember to manually null or reassign them after use.

### Hanging outer-function variables in closures

A closure is a function that is nested within a function—in other words, an inner or enclosed function. Closures have access to the variables and scope of the outer function. Similar to accidental global variables, it’s possible for a function declared in the outer scope to still be residing in memory after the outer function has executed because some inner function still has access but isn’t using it.

### Detached DOM/Out of DOM references

The document object model (DOM) is a doubly-linked tree in which any reference to any node in the tree will prevent the entire tree from garbage collection. A detached DOM occurs when a node is removed from the tree but is still retained in memory by a reference within JavaScript. If you don’t handle the reference in JavaScript, the garbage collector will not sweep the reference and your code will continue to consume memory.

## **Limit variable calls:**

Declaring variables and calling back on them for references is the bread and butter of coding. But as we discussed previously everytime JavaScript holds a reference to a variable, memory is consumed, and the potential for memory leaks increases. As we already covered above, accidental global variables, hanging closures, and Out of DOM references can all have the potential to impact performance via memory leaks.

Simply limiting the number of variable calls can lead to writing more concise performant code. If you find yourself declaring too many variables, consider if there is a better way to write your function to achieve the same result with better performance.

## **Test your code:**

Measurement is the key to improvement. And it’s by testing your code that you can identify performance issues such as memory leaks and patch them. Here are some popular JavaScript testing tools:

* Console.time()

[Console.time()](https://developer.mozilla.org/en-US/docs/Web/API/Console/time) is a simple built-in JavaScript function that you can use to track how long an operation takes.

Operation time will be tracked from start to end giving you the effective process time.