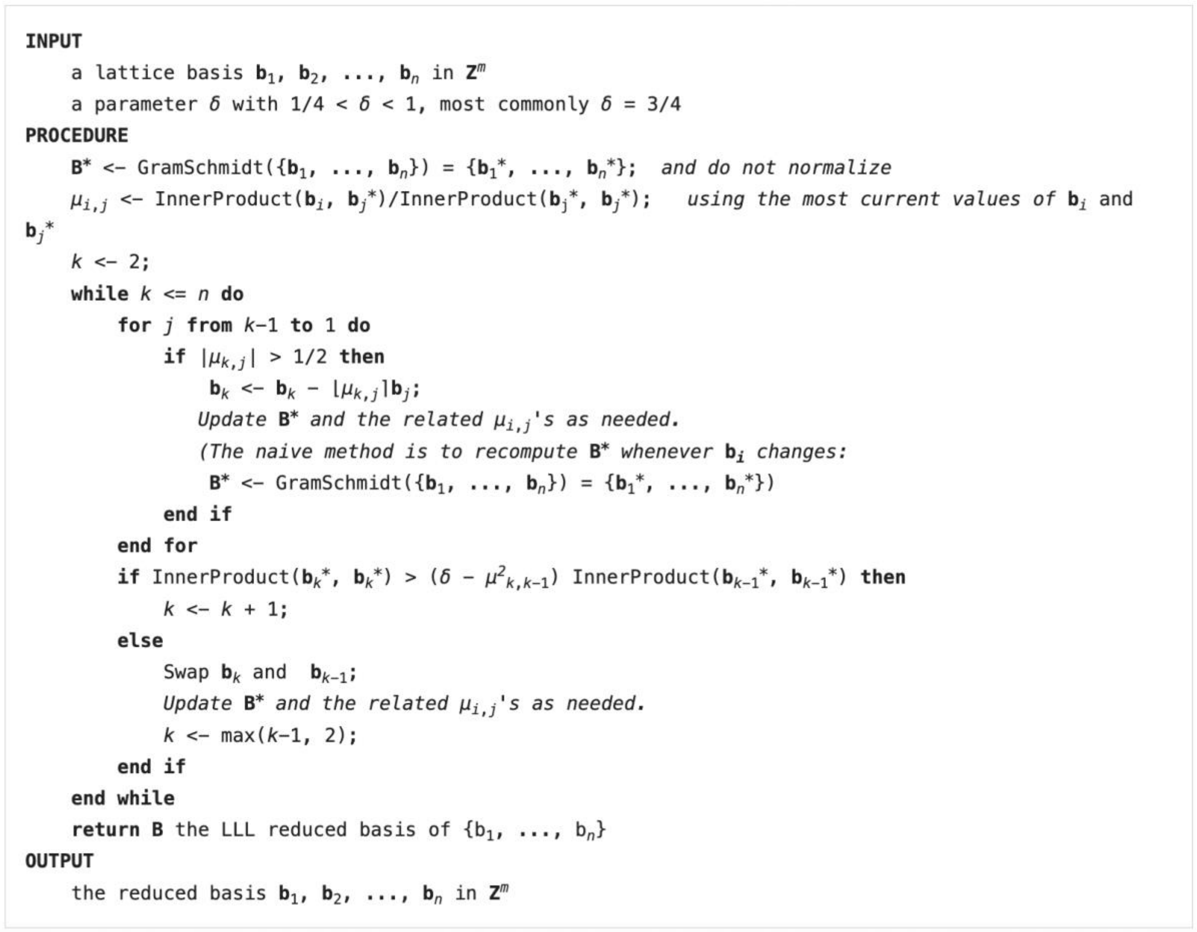
Programming Paradigm Summative

Problem Definition:
The Shortest Vector Problem is a fundamental problem in lattice-based cryptography, with various real-world applications. The assignment requires us to find the shortest non-zero vector v ∈ L such that the Euclidian norm (L^2) of v is minimised.

My Implementation:

My solution implements 3 components. I used the Gram-Schmidt Process to orthogonalize all input basis vectors followed by the Lenstra–Lenstra–Lovász lattice basis reduction algorithm to further reduce the basis and finally, I used a brute force approach to enumerate all possible vector combinations.

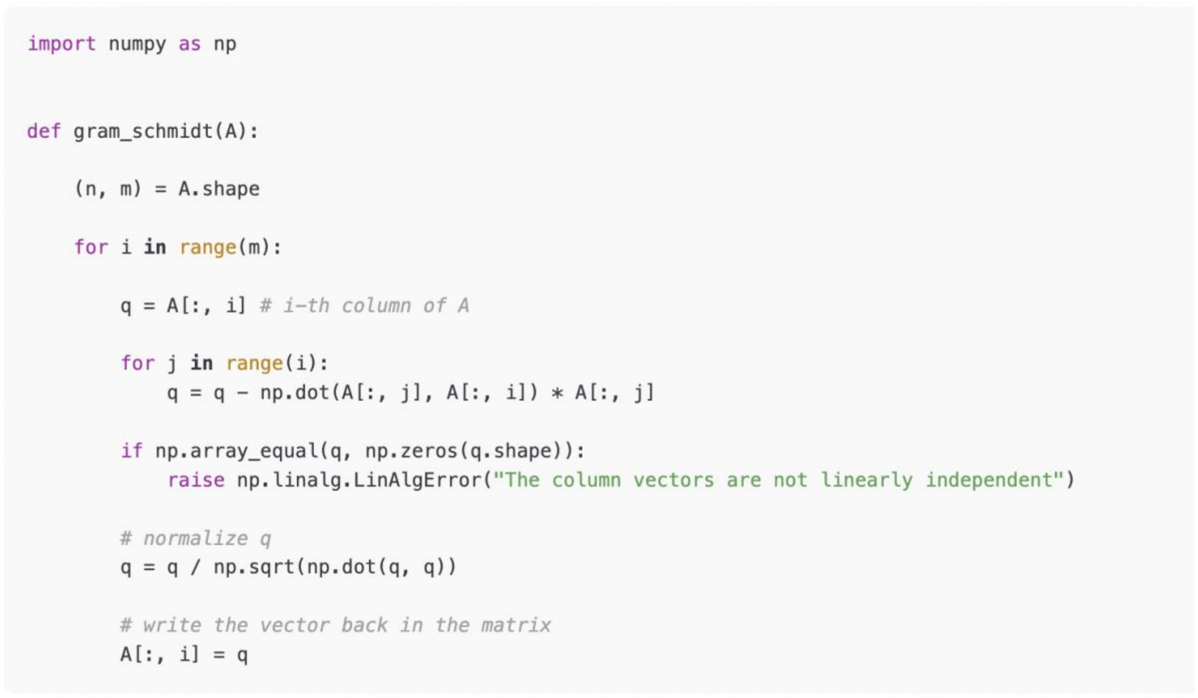
I first used pseudocode to visualize how to approach creating the solution.



```
INPUT
    a lattice basis $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ in $\mathbf{Z}^m$
    a parameter $\delta$ with $1/4 < \delta < 1$, most commonly $\delta = 3/4$
PROCEDURE
    $\mathbf{B}^*$ <- GramSchmidt({$\mathbf{b}_1, \ldots, \mathbf{b}_n$}) = {$\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*$};  and do not normalize
    $\mu_{i,j}$ <- InnerProduct($\mathbf{b}_i$, $\mathbf{b}_j^*$)/InnerProduct($\mathbf{b}_j^*$, $\mathbf{b}_j^*$);   using the most current values of $\mathbf{b}_i$ and
$\mathbf{b}_j^*$
    $k$ <- 2;
    while $k$ <= $n$ do
        for $j$ from $k$-1 to 1 do
            if $|\mu_{k,j}| > 1/2$ then
                $\mathbf{b}_k$ <- $\mathbf{b}_k - \lfloor\mu_{k,j}\rceil\mathbf{b}_j$;
               Update $\mathbf{B}^*$ and the related $\mu_{i,j}$'s as needed.
               (The naive method is to recompute $\mathbf{B}^*$ whenever $\mathbf{b}_i$ changes:
                $\mathbf{B}^*$ <- GramSchmidt({$\mathbf{b}_1, \ldots, \mathbf{b}_n$}) = {$\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*$})
            end if
        end for
        if InnerProduct($\mathbf{b}_k^*$, $\mathbf{b}_k^*$) > ($\delta - \mu^2_{k,k-1}$) InnerProduct($\mathbf{b}_{k-1}^*$, $\mathbf{b}_{k-1}^*$) then
            $k$ <- $k$ + 1;
        else
            Swap $\mathbf{b}_k$ and  $\mathbf{b}_{k-1}$;
            Update $\mathbf{B}^*$ and the related $\mu_{i,j}$'s as needed.
            $k$ <- max($k$-1, 2);
        end if
    end while
    return B the LLL reduced basis of {$b_1, \ldots, b_n$}
OUTPUT
    the reduced basis $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ in $\mathbf{Z}^m$
```

Figure 1: Visualisation of how to implement LLL reduction (HOFFSTEIN, 2008)

I then referred to a source I found online (Zerobone, 2021) on how to do Gram-Schmidt process.



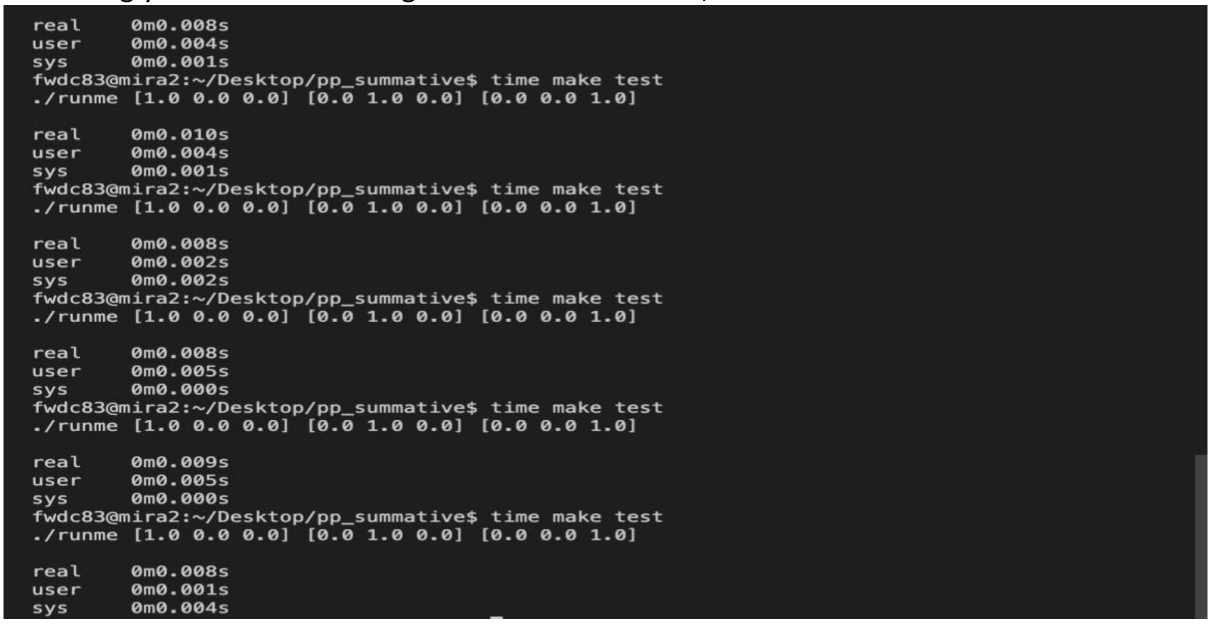
```
import numpy as np


def gram_schmidt(A):

    (n, m) = A.shape

    for i in range(m):

        q = A[:, i] # i-th column of A

        for j in range(i):
            q = q - np.dot(A[:, j], A[:, i]) * A[:, j]

        if np.array_equal(q, np.zeros(q.shape)):
            raise np.linalg.LinAlgError("The column vectors are not linearly independent")

        # normalize q
        q = q / np.sqrt(np.dot(q, q))

        # write the vector back in the matrix
        A[:, i] = q
```

Figure 2: Python Implementation of Gram-Schmidt Process

I then employed a brute force method to iterate through all possible vector coefficients for each dimension and choosing the minimum length based on the output of the LLL algorithm.

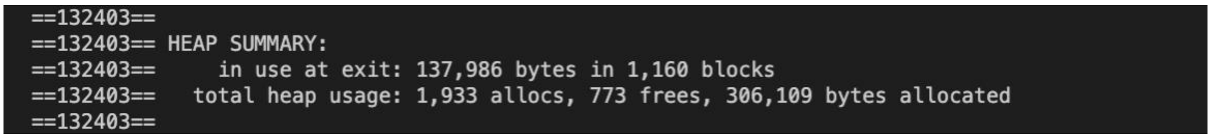
Successes:

The implementation of test case './runme [1.0 0.0 0.0] [0.0 1.0 0.0] [0.0 0.0 1.0]' worked exceedingly well with an average runtime of 0.00872s, taken over 50 occurrences.



```
real    0m0.008s
user    0m0.004s
sys     0m0.001s
fwdc83@mira2:~/Desktop/pp_summative$ time make test
./runme [1.0 0.0 0.0] [0.0 1.0 0.0] [0.0 0.0 1.0]

real    0m0.010s
user    0m0.004s
sys     0m0.001s
fwdc83@mira2:~/Desktop/pp_summative$ time make test
./runme [1.0 0.0 0.0] [0.0 1.0 0.0] [0.0 0.0 1.0]

real    0m0.008s
user    0m0.002s
sys     0m0.002s
fwdc83@mira2:~/Desktop/pp_summative$ time make test
./runme [1.0 0.0 0.0] [0.0 1.0 0.0] [0.0 0.0 1.0]

real    0m0.008s
user    0m0.005s
sys     0m0.000s
fwdc83@mira2:~/Desktop/pp_summative$ time make test
./runme [1.0 0.0 0.0] [0.0 1.0 0.0] [0.0 0.0 1.0]

real    0m0.009s
user    0m0.005s
sys     0m0.000s
fwdc83@mira2:~/Desktop/pp_summative$ time make test
./runme [1.0 0.0 0.0] [0.0 1.0 0.0] [0.0 0.0 1.0]

real    0m0.008s
user    0m0.001s
sys     0m0.004s
```

The total memory usage of solution is as follows:



```
==132403==
==132403== HEAP SUMMARY:
==132403==     in use at exit: 137,986 bytes in 1,160 blocks
==132403==   total heap usage: 1,933 allocs, 773 frees, 306,109 bytes allocated
==132403==
```

Constraints during and after:
The implementation of the './runme [1.0 0.0 0.0] [0.0 1.0 0.0] [0.0 0.0 1.0]' test case was difficult at first due to the way my matrix parsing function took [1.0 and 0.0 as separate arguments instead of [1.0 0.0 0.0]. I had to brainstorm in order to save the format in a way that the function can be implemented correctly.

While my algorithm may work well with inputs of low dimensionality, the runtime and memory usage of my solution goes up exponentially as more inputs are added. This makes it highly inefficient for high-dimensionality basis as exponential growth is generally considered undesirable for large input sizes because it leads to impractical runtimes and resource requirements. Efficient algorithms should ideally have polynomial or sub-exponential time complexity to handle larger instances of the problem more effectively.

Conclusion:
Although the implementation of the solution does work, which works well with low-dimensional lattices, the constraints of LLL and Grand-Schmidt while accurate, the time complexity of both processes contributes an exponential growth in both run time and memory usage. The exponential increase in resource requirements indicate that this algorithm will be increasingly unsuitable to use as dimensionality increases which can lead to impractical runtimes and memory demands. Thus , the solution while effective in certain scenarios (i.e low-dimensionality), highlights the importance on ensuring the algorithm is scalable with higher-dimensions.

References:

HOFFSTEIN, J. (2008). *Introduction to mathematical cryptography*. SPRINGER-VERLAG NEW YORK.

Zerobone. (2021, February 18). *Implementing and visualizing gram-schmidt orthogonalization*. ZeroBone. https://zerobone.net/blog/cs/gram-schmidt-orthogonalization/