# Programming Paradigms Report
# Aleksandar Yordanov.

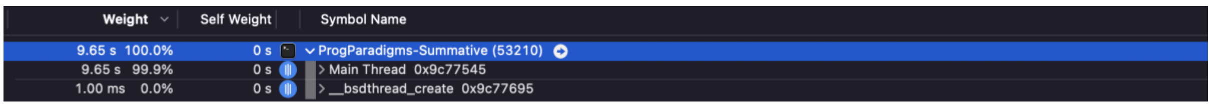
## Lattice iteration:

My approach to finding the shortest vector is by using lattice iteration. I take every single integer linear combination of each basis vector within a given radius. I.e., for a radius of 3, it would take every linear combination from coefficient -3 to 3. This gives me each lattice point around the centre point. For each lattice point, I then take its Euclidian norm and compare it to the smallest distance recorded. If it is smaller, then I store it.

I implement this by generating an array of coefficients to use for my linear combinations and iterate each coefficient from -r to r. I then calculate each component of each lattice point based on this array and add them to a centre point vector. Finally, I calculate the Euclidian norm and compare to the smallest recorded distance.

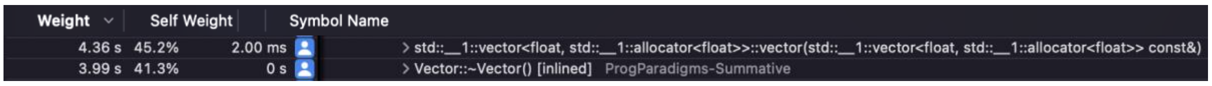
## Optimisation:

When programming my solution, to initially optimise, I passed as many objects as possible by reference to avoid any copying, which adds some overhead.  Once I had completed my first naïve implementation. The first thing I did to optimise my program was using the profiler built into XCode to obtain a stack trace:



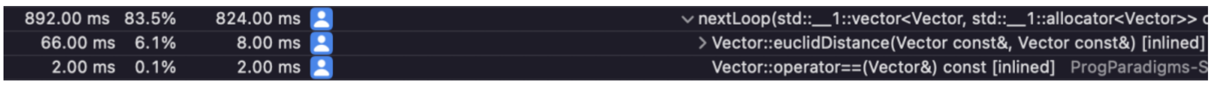
```
        Weight         Self Weight   Symbol Name
9.65 s    100.0%       0 s           ⌄ ProgParadigms-Summative (53210)
9.65 s    99.9%        0 s             > Main Thread  0x9c77545
1.00 ms   0.0%         0 s             > __bsdthread_create  0x9c77695
```

For a dimension of 10 and a radius of 2. We can see that it took 9.65 seconds using std::vector to find the shortest vector. So, I took a look at the call tree to see what was taking up the most time.



```
Weight           Self Weight   Symbol Name
4.36 s   45.2%   2.00 ms       > std::__1::vector<float, std::__1::allocator<float>>::vector(std::__1::vector<float, std::__1::allocator<float>> const&)
3.99 s   41.3%   0 s           > Vector::~Vector() [inlined]   ProgParadigms-Summative
```

From this we can see that 86.5% of the time, my code was either allocating memory or freeing memory. To solve this issue, I decided to implement arrays instead of vectors. However, you cannot set the size of an array on the stack at runtime, so to work around this, I created another program to take the users input, process it to get dimension and data, write it to headers as a #define and compile my solution. This provided a massive increase in performance and allowed me to use zero heap allocations.



```
892.00 ms   83.5%   824.00 ms   ⌄ nextLoop(std::__1::vector<Vector, std::__1::allocator<Vector>> c
 66.00 ms    6.1%     8.00 ms     > Vector::euclidDistance(Vector const&, Vector const&) [inlined]
  2.00 ms    0.1%     2.00 ms       Vector::operator==(Vector&) const [inlined]   ProgParadigms-S
```

The other largest optimisation that I implemented was pruning the search space to avoid symmetrical points, which have the same Euclidian norm. I achieved this by only checking diagonally in the vector space:

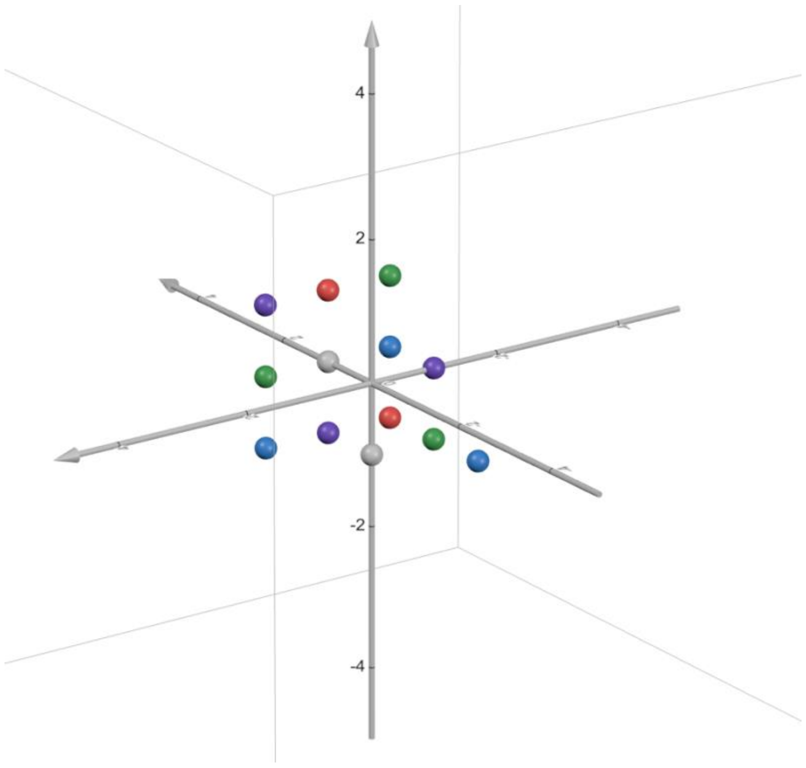
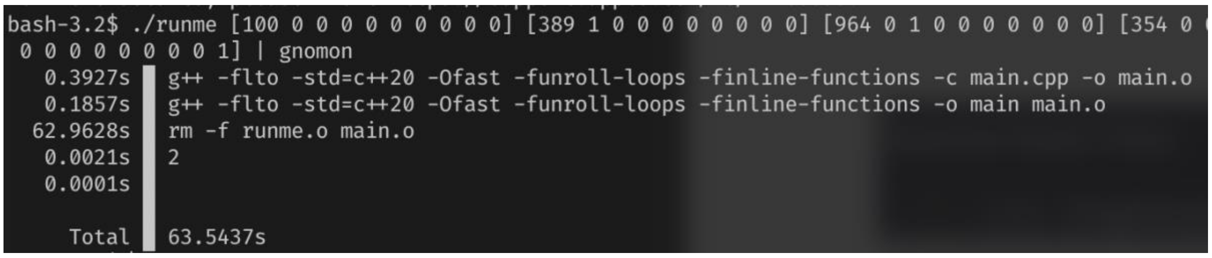


*Figure 1 (Example in 3D, excluding centre point)*

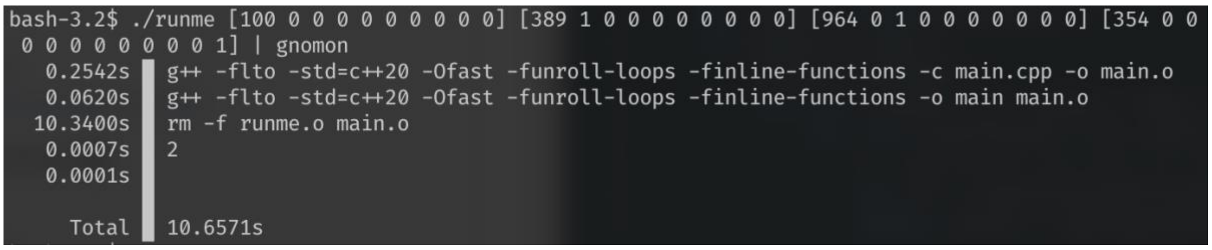
This greatly reduces the number of points needed to check and resulted in an 80% reduction in processing time:



```
bash-3.2$ ./runme [100 0 0 0 0 0 0 0 0 0 0] [389 1 0 0 0 0 0 0 0 0 0] [964 0 1 0 0 0 0 0 0 0 0] [354 0
 0 0 0 0 0 0 0 0 1] | gnomon
   0.3927s ▌ g++ -flto -std=c++20 -Ofast -funroll-loops -finline-functions -c main.cpp -o main.o
   0.1857s ▌ g++ -flto -std=c++20 -Ofast -funroll-loops -finline-functions -o main main.o
  62.9628s ▌ rm -f runme.o main.o
   0.0021s ▌ 2
   0.0001s ▌

     Total ▌ 63.5437s
```

*Figure 2 (Before, r = 4)*

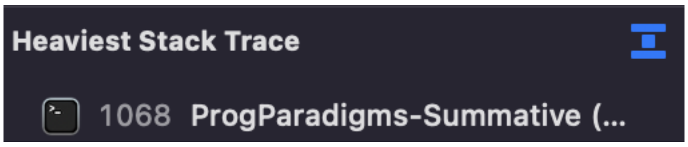


```
bash-3.2$ ./runme [100 0 0 0 0 0 0 0 0 0 0] [389 1 0 0 0 0 0 0 0 0 0] [964 0 1 0 0 0 0 0 0 0 0] [354 0 0
 0 0 0 0 0 0 0 0 1] | gnomon
   0.2542s ▌ g++ -flto -std=c++20 -Ofast -funroll-loops -finline-functions -c main.cpp -o main.o
   0.0620s ▌ g++ -flto -std=c++20 -Ofast -funroll-loops -finline-functions -o main main.o
  10.3400s ▌ rm -f runme.o main.o
   0.0007s ▌ 2
   0.0001s ▌

     Total ▌ 10.6571s
```

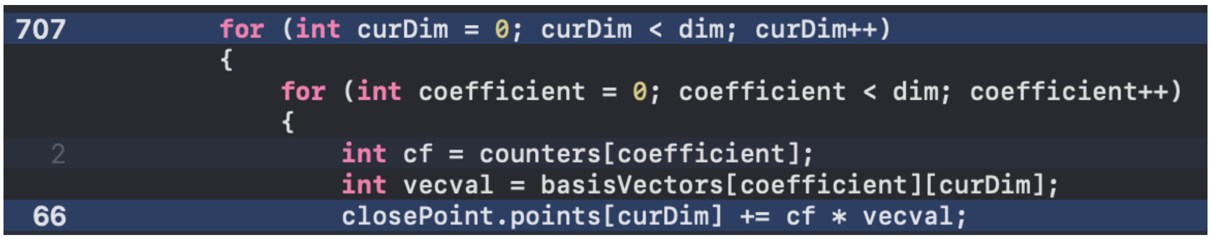
*Figure 3 (After, r = 4)*

Another optimisation I implemented was contiguous memory access:



Heaviest Stack Trace

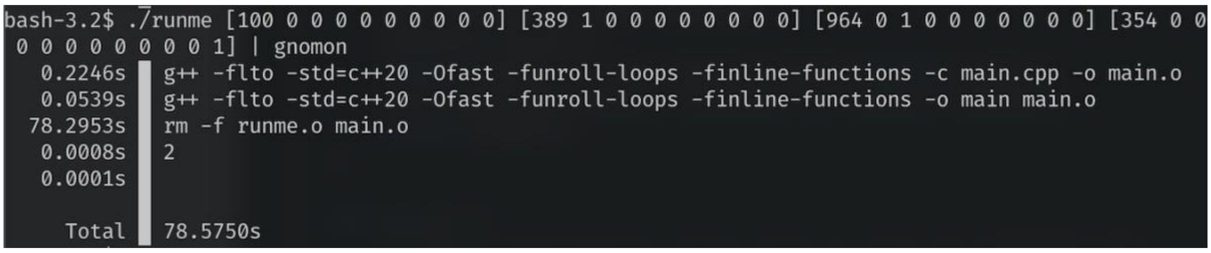
1068 ProgParadigms-Summative (...



```
707         for (int curDim = 0; curDim < dim; curDim++)
            {
                for (int coefficient = 0; coefficient < dim; coefficient++)
                {
  2                 int cf = counters[coefficient];
                    int vecval = basisVectors[coefficient][curDim];
 66                 closePoint.points[curDim] += cf * vecval;
```

We can see that out of 1068 samples, 773 are in the loop where I calculate the linear combination based on the coefficients in counters. On the second to last line, I am accessing basisVectors by column which is non-contiguous memory access as I am accessing every dim'th element.

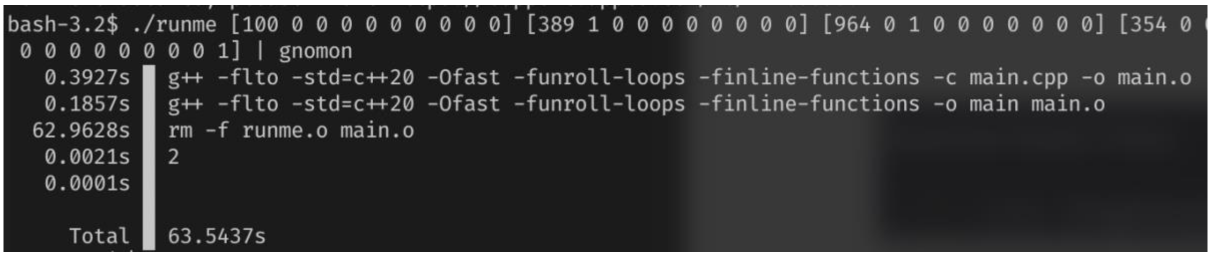
The optimisation I implemented to remedy this was flattening and swapping the rows and columns of the basis matrix. That way memory access is contiguous. With contiguous memory, elements are next to each other, minimising cache misses. This allows for compiler auto-vectorisation to yield some performance benefits. This optimisation reduced my execution time by around 20%:



```
bash-3.2$ ./runme [100 0 0 0 0 0 0 0 0 0 0] [389 1 0 0 0 0 0 0 0 0 0] [964 0 1 0 0 0 0 0 0 0 0] [354 0 0
 0 0 0 0 0 0 0 0 1] | gnomon
   0.2246s ▌ g++ -flto -std=c++20 -Ofast -funroll-loops -finline-functions -c main.cpp -o main.o
   0.0539s ▌ g++ -flto -std=c++20 -Ofast -funroll-loops -finline-functions -o main main.o
  78.2953s ▌ rm -f runme.o main.o
   0.0008s ▌ 2
   0.0001s ▌

     Total ▌ 78.5750s
```

*Figure 4 (Before, r = 4)*



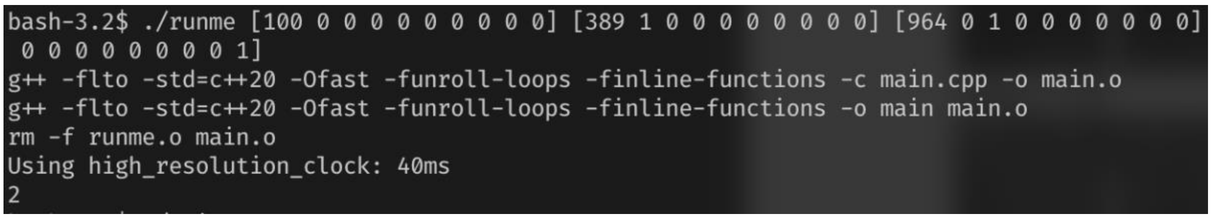
```
bash-3.2$ ./runme [100 0 0 0 0 0 0 0 0 0 0] [389 1 0 0 0 0 0 0 0 0 0] [964 0 1 0 0 0 0 0 0 0 0] [354 0
 0 0 0 0 0 0 0 0 1] | gnomon
   0.3927s ▌ g++ -flto -std=c++20 -Ofast -funroll-loops -finline-functions -c main.cpp -o main.o
   0.1857s ▌ g++ -flto -std=c++20 -Ofast -funroll-loops -finline-functions -o main main.o
  62.9628s ▌ rm -f runme.o main.o
   0.0021s ▌ 2
   0.0001s ▌

     Total ▌ 63.5437s
```

*Figure 5 (After, r = 4)*

Other optimisations I implemented were algorithmic optimisations such as returning early when a point that is not zero is found and fixing small bugs in other algorithms.

After implementing every optimisation and comparing back to the naïve solution:

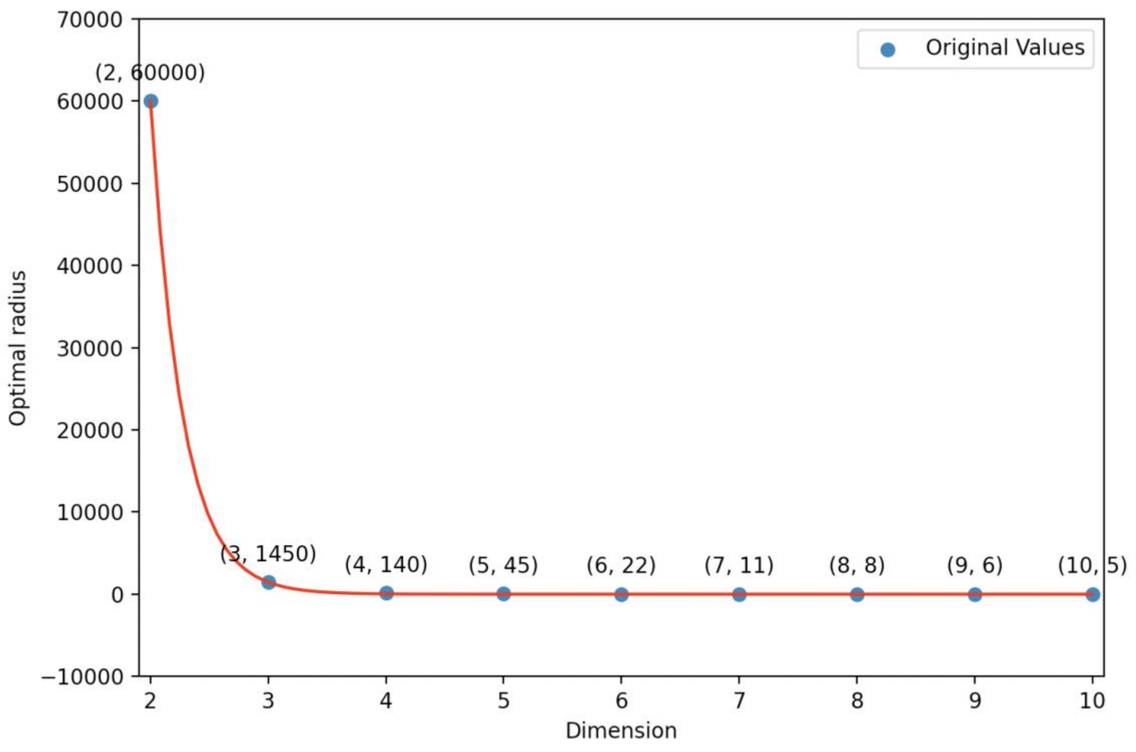


```
bash-3.2$ ./runme [100 0 0 0 0 0 0 0 0 0] [389 1 0 0 0 0 0 0 0 0] [964 0 1 0 0 0 0 0 0 0]
 0 0 0 0 0 0 0 0 0 1]
g++ -flto -std=c++20 -Ofast -funroll-loops -finline-functions -c main.cpp -o main.o
g++ -flto -std=c++20 -Ofast -funroll-loops -finline-functions -o main main.o
rm -f runme.o main.o
Using high_resolution_clock: 40ms
2
```

I reduced the processing time for dim=10 and r=2 from 9.65 seconds to 40ms resulting in a 240x improvement.

## Optimal radius and time complexity:

The problem with brute-forcing is that with more complex vectors, the higher the search area needs to be.



This graph shows the radius for which the shortest vector is found in approximately 60 seconds. We can see that the radius goes down super-exponentially depending on the dimension. This can also be inferred from the fact that my solution has n=dim nested loops where each iterates over the value of the counters leading to O((2 * radius + 1)^dim).

However, one of the benefits of brute-forcing is memory complexity. Because everything in nextLoop is either a local variable or a reference. The only factors that contribute to space complexity are the counter array and the basis vectors. Leading to a space complexity of O(dim).

## Final words:

I understood that I could have implemented LLL quite easily to reduce the complexity of the basis vectors and save time, however I wanted to challenge myself and see how far I could optimise a brute-force algorithm and not using LLL allowed me to visualise the difference in processing time with each step.