```
import numpy as np
import random

def faster_matrix_product(mat1, mat2):
    assert mat1.shape[1] == mat2.shape[0]
    result = []
    for r in range(mat1.shape[0]):
        row = []
        for c in range(mat2.shape[1]):
            value = np.dot(mat1[r],mat2.T[c])
            row.append(value)
        result.append(row)
    return np.array(result)

list1 = np.array([2,3,4,5])
a = random.choice(list1)
matrix1 = np.random.rand(a, a)
matrix2 = np.random.rand(a, a)

assert np.allclose(matrix1 @ matrix2, faster_matrix_product(matrix1, matrix2))
print(faster_matrix_product(matrix1, matrix2))
print(matrix1 @ matrix2)
```

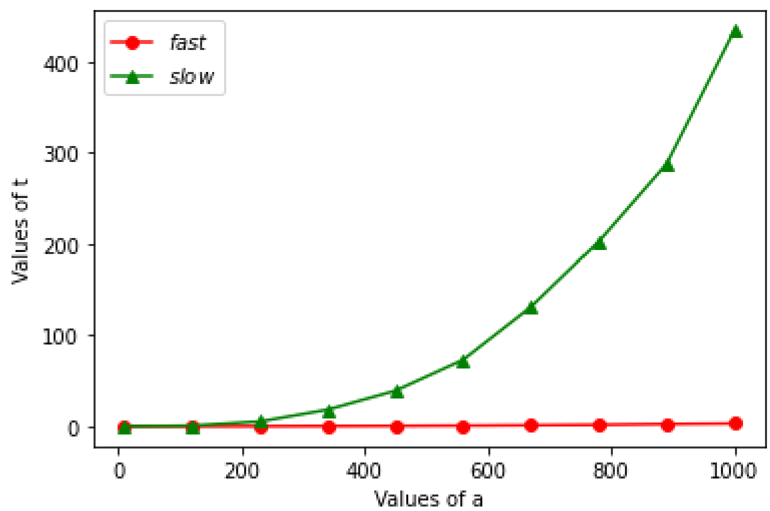
```
[[0.40950349 0.52169498 0.26955713 0.22566952]
 [1.39294954 1.31235593 0.59989205 0.70335578]
 [1.48730313 1.63687245 1.10793999 0.76734368]
 [1.49842777 1.30032915 1.01184722 0.58568765]]
[[0.40950349 0.52169498 0.26955713 0.22566952]
 [1.39294954 1.31235593 0.59989205 0.70335578]
 [1.48730313 1.63687245 1.10793999 0.76734368]
 [1.49842777 1.30032915 1.01184722 0.58568765]]
```

1. **REASON 1** : This function only uses "for" loop once, which means it has less code and is easy to store.
2. **REASON 2** : Time complexity of this function is n^2 which is much less than n^3, which means it is more efficient and cost less time when running.

```
import numpy as np
from timeit import timeit
import matplotlib.pylab as plt

def faster_matrix_product(mat1, mat2):
    assert mat1.shape[1] == mat2.shape[0]
    result = []
    for r in range(mat1.shape[0]):
        row = []
        for c in range(mat2.shape[1]):
            value = np.dot(mat1[r],mat2.T[c])
            row.append(value)
        result.append(row)
    return np.array(result)

def slow_matrix_product(mat1, mat2):
```

```
        assert mat1.shape[1] == mat2.shape[0]

        result = []
        for c in range(mat2.shape[1]):
                column = []
                for r in range(mat1.shape[0]):
                        value = 0
                        for i in range(mat1.shape[1]):
                                value += mat1[r, i] * mat2[i, c]
                        column.append(value)
                result.append(column)
        return np.array(result).transpose()


Y1 = []
Y2 = []
c = np.linspace(10,1000,10, dtype=int)
for a in c:
    mat1 = np.random.rand(a, a)
    mat2 = np.random.rand(a, a)
    Y1.append(timeit(lambda: faster_matrix_product(mat1, mat2),number=1))
    Y2.append(timeit(lambda: slow_matrix_product(mat1, mat2),number=1))

plt.plot(c, Y1, "ro-")
plt.plot(c, Y2, "g^-")
plt.xlabel("Values of a")
plt.ylabel("Values of t")
plt.legend(["$fast$","$slow$"])
```

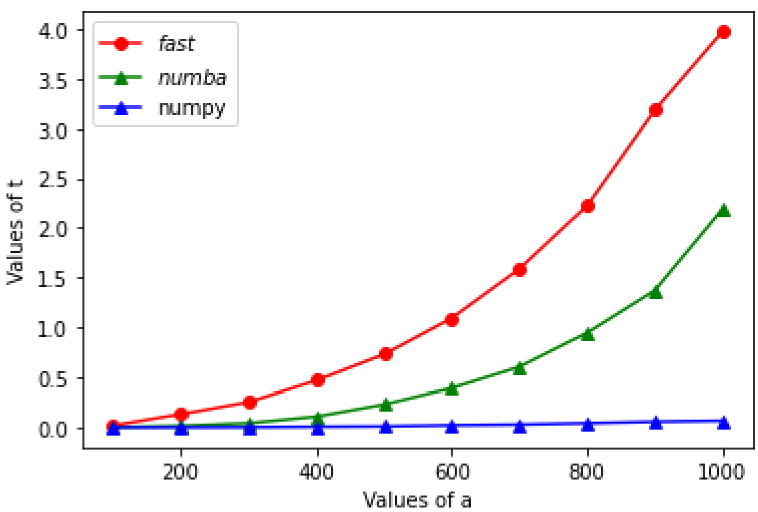
```
<matplotlib.legend.Legend at 0x7fa22fc17890>
```




```
import numpy as np
from timeit import timeit
import matplotlib.pylab as plt
from numba import njit, jit

def faster_matrix_product(mat1, mat2):
        assert mat1.shape[1] == mat2.shape[0]
        result = []
        for r in range(mat1.shape[0]):
            row = []
            for c in range(mat2.shape[1]):
```

```
                value = np.dot(mat1[r],mat2.T[c])
                row.append(value)
            result.append(row)
        return np.array(result)

@njit
def faster_matrix_product_nb(mat1, mat2):
        assert mat1.shape[1] == mat2.shape[0]
        result = []
        for r in range(mat1.shape[0]):
            row = []
            for c in range(mat2.shape[1]):
                value = np.dot(mat1[r],mat2.T[c])
                row.append(value)
            result.append(row)
        return np.array(result)

def numpy_product(mat1, mat2):
        return np.array(mat1@mat2)

Y1 = []
Y2 = []
Y3 = []
c = np.linspace(2,1000,11, dtype=int)
for a in c:
    mat1 = np.random.rand(a, a)
    mat2 = np.random.rand(a, a)
    Y1.append(timeit(lambda: faster_matrix_product(mat1, mat2),number=1))
    Y2.append(timeit(lambda: faster_matrix_product_nb(mat1, mat2),number=1))
    Y3.append(timeit(lambda: numpy_product(mat1, mat2),number=1))

# c[0] is used to initialize numba
plt.plot(c[1:], Y1[1:], ”ro-”)
plt.plot(c[1:], Y2[1:], ”g^-”)
plt.plot(c[1:], Y3[1:], ”b^-”)
plt.xlabel(”Values of a”)
plt.ylabel(”Values of t”)
plt.legend([”$fast$”,”$numba$”,”numpy”])
```

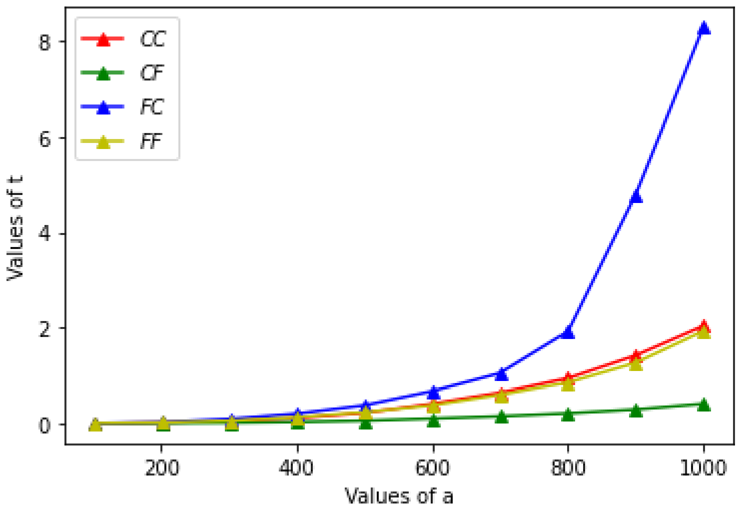
```
<matplotlib.legend.Legend at 0x7f8dc8bc4150>
```




```
import numpy as np
```

```
from timeit import timeit
import matplotlib.pylab as plt
from numba import njit, jit

@njit
def faster_matrix_product(mat1, mat2):
        assert mat1.shape[1] == mat2.shape[0]
        result = []
        for r in range(mat1.shape[0]):
            row = []
            for c in range(mat2.shape[1]):
                value = np.dot(mat1[r],mat2.T[c])
                row.append(value)
            result.append(row)
        return np.array(result)

C_C = []
C_Fortran = []
Fortran_C = []
Fortran_F = []
c = np.linspace(2,1000,11, dtype=int)

for a in c:
    mat1 = np.random.rand(a, a)
    mat2 = np.random.rand(a, a)
    mat1_F = np.asfortranarray(mat1)
    mat2_F = np.asfortranarray(mat2)
    C_C.append(timeit(lambda: faster_matrix_product(mat1, mat2),number=1))
    C_Fortran.append(timeit(lambda: faster_matrix_product(mat1, mat2_F),number=1))
    Fortran_C.append(timeit(lambda: faster_matrix_product(mat1_F, mat2),number=1))
    Fortran_F.append(timeit(lambda: faster_matrix_product(mat1_F, mat2_F),number=1))

# c[0] is used to initialize numba
plt.plot(c[1:], C_C[1:], "r^-")
plt.plot(c[1:], C_Fortran[1:], "g^-")
plt.plot(c[1:], Fortran_C[1:], "b^-")
plt.plot(c[1:], Fortran_F[1:], "y^-")
plt.xlabel("Values of a")
plt.ylabel("Values of t")
plt.legend(["$CC$","$CF$","$FC$","$FF$"])
```

```
<matplotlib.legend.Legend at 0x7f8dc931f110>
```



REASON : It will be the fastest when the first input is C-style and the second is Fortan-style because in this function we calculate the product of the row of matrix1 and the column of matrix2, then we only need to access continuous memory in each loop because C-style stores row of matrix continuously and Fortran-style stores column of matrix continuously, which will be more efficient and faster than others.