1.1 Fibonacci

1.1.1 The python code for this function is:

```
def fibonacci(n):
    if n == 1:
        return 1
    elif n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

1.1.2 The complexity of this implementation is $O(2^n)$. Since the implementation is using recursion, so the time complexity $T(n)$ satisfies:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Which is exponential.

1.1.3 The python code of the alternative implementation is:

```
def fibonacci2(n):
    if n == 1:
        return 1
    elif n == 2:
        return 1
    else:
        f1 = 1
        f2 = 1
        for i in range(3, n + 1):
            f = f1 + f2
            f1 = f2
            f2 = f
    return f2
```

1.1.4 The complexity of this implementation is $O(n)$. Since we use a for loop in our implementation, and it just goes through $3{\sim}n$. With some extra $O(1)$ operations, the final time complexity will be $O(n)$

1.1.5 There are some examples for computational performance improvement:

a. Avoid repeating computation, we can store computed values for future use but not recalculate them. This strategy can be also called trading time with space. E.g, dynamic programming.

b. Divide and conquer. We can divide big problem into same but smaller problems and solve smaller problems. E.g, merge sort, quick sort.

c. Parallelization. We can use multiprocessor to solve complicate problems.

2.1 Finding π in a random uniform.

We can achieve this by picking lots of random coordinates in an x-y grid and calculating if they are within the circle or the square.

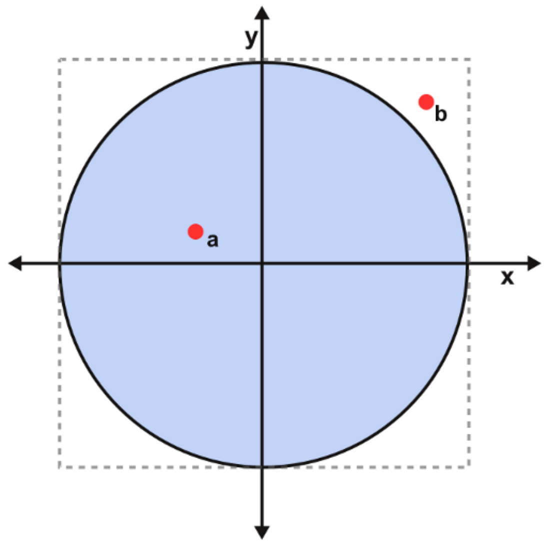

Fig 1.

By assigning the radius to be 1, for a random generated point (x, y), if $x^2 + y^2 \leq 1$ then the point lies inside the circle's radius. In Fig 1 we see the point a is in the circle while point b lies outside the circle.

If we generate N random points, and we count there are M points are inside the circle. Then we have:

$$\frac{M}{N} = \frac{Area\ of\ circle}{Area\ of\ square} = \frac{\pi}{4}$$

So we can get the approximate $\pi = 4\frac{M}{N}$

Then we can write following python code:

```
def compute_pi():
    total = 10000000
    count = 0
    for i in range(total):
        x = random.uniform(-1,1)
        y = random.uniform(-1,1)
        if x*x+y*y <= 1.0:
            count += 1
    return float(count) / total * 4
```

And I have also computed a table of our approximate π and N:

N	10000	100000	500000	1000000	5000000	10000000	100000000
π	3.1208	3.14812	3.143568	3.14068	3.1422768	3.1419388	3.14170752