CS 312

Tyler Blackham

Project 1 Report

Space & Time Complexity

```python
def mod_exp(x: int, y: int, N: int) -> int:
    if y == 0:
        return 1
    z = mod_exp(x, y//2, N)
    if y % 2 == 0:
        return (z**2) % N
    else:
        return (x * z**2) % N
```

The Big O Complexity of mod_exp is O(n^3) because multiplication is O(n^2) and it has to do n multiplies (n * n^2 = n^3)

The Big O Complexity of fermat is also O(n^3), assuming that the complexity of randint is less than or equal to mod_exp which is O(n^3). Mod_exp gets called k times but the constant can be disregarded.

```python
def fermat(N: int, k: int) -> str:
    for i in range(k):
        a: int = random.randint(1, N-1)
        if mod_exp(a, N-1, N) != 1:
            return "composite"
    return "prime"
```

```python
def miller_rabin(N: int, k: int) -> str:
    for i in range(k):
        a = random.randint(1, N-1)
        if mod_exp(a, N - 1, N) != 1:
            return "composite"
        new_N:int = N
        while True:
            if new_N % 2 == 1:
                break
            new_N = new_N//2
            exp_result = mod_exp(a, new_N-1, N)
            if exp_result == -1:
                break
            if exp_result != 1:
                return "composite"
    return "prime"
```

The time complexity of miller_rabin is O((n^3)*log(n)) because, like fermat, the most complex part is calling mod_exp. However, unlike fermat, this algorithm has the potential to call mod_exp a maximum of log n times, if N is prime.

Generate_large_prime calls fermat which O(n^3). Because there is roughly a 1/n chance of the random number being prime, it will take, on average, n rounds before getting a prime number. So the time complexity is O(n^4).

```python
def generate_large_prime(bits=512) -> int:  3 usages
    """
    Generate a random prime number with the specified bit length.
    Use random.getrandbits(bits) to generate a random number of the
     specified bit length.
    """
    while True:
        x = random.getrandbits(bits)
        k = 20
        if fermat(x, k) == "prime":
            return x
```

The time complexity of ext_euclid is O(n^3) because the most complex call is the multiplication and division which are both O(n^2) and it has to repeat itself n times (n * n^2 = n^3).

```python
def ext_euclid(a: int, b: int) -> tuple[int, int, int]:
    """
    The Extended Euclid algorithm
    Returns x, y , d such that:
    - d = GCD(a, b)
    - ax + by = d

    Note: a must be greater than b
    """
    if b == 0:
        return 1, 0, a
    x, y, d = ext_euclid(b, a % b)
    return y, x - (a//b)*y, d
```

Generate_key_pairs uses everything else. Generate_large_prime is O(n^4) and although it might repeat, there is a 1/(2^n) chance of that happening, so it's pretty negligible. The multiplications are O(n^2) so they can be disregarded. The for each loop happens a constant number of times because the size of primes is not reliant on n. However, ext_euclid is also O(n^3) but it can also be ignored since it is less complex than O(n^4). All in all, generate_key_pairs is of the time complexity O(n^4).

```python
def generate_key_pairs(bits: int) -> tuple[int, int, int]:  2 usages
    """
    Generate RSA public and private key pairs.
    Return N, e, d
    - N must be the product of two random prime numbers p and q
    - e and d must be multiplicative inverses mod (p-1)(q-1)
    """
    p = generate_large_prime(bits)
    q = generate_large_prime(bits)
    while q == p:
        q = generate_large_prime(bits)
    N = p * q
    i = (p-1)*(q-1)
    e = 0
    d = 0
    for prime in primes:
        x, y, gcd = ext_euclid(i, prime)
        if gcd == 1:
            e = prime
            d = y
            break
    if d < 0:
        d = i + d
    return N, e, d
```

Correctness Probability

Fermat

```python
def fprobability(k: int) -> float:
    return 1 - (1/(2**k))
```

I chose 1 - (1/(2^k)) as the probability out of one that the fermat algorithm is correct because there is a 1/2 chance that the random “a” returns a 1 when it is modularly exponentiated. Every time you repeat it with a different “a”, that chance goes down to (1/2)^k. Thus, subtracting that small chance from one is the chance that it is correct.

Miller Rabin

```python
def mprobability(k: int) -> float:
    return 1 - (1/(4**k))
```

Similarly to fermat, since the probability of the random “a” returning a 1 is 1/4, the probability of it decreases to (1/4)^k. Thus, the probability that it is correct is 1 - (1 / (4^k)) out of 1.