1.

In the RSA algorithm, it is not necessary for gcd(*m*,*n*)=1. Here, *m* represents the message to be encrypted, and *n* is the modulus, a product of two large primes *p* and *q*. The requirement for RSA to work correctly is that *m* should be in the range 0≤*m*<*n*.

However, during the key generation phase, it is crucial that the chosen encryption exponent 𝑒 and *ϕ*(*n*) (where *ϕ*(*n*)=(*p*−1)(*q*−1)) are coprime, i.e., gcd(*e*,*ϕ*(*n*))=1. This ensures that 𝑒*e* has an inverse *d*, which is used for decryption.

Here is a short proof:

The RSA cryptosystem will still work if 𝑚 shares a common factor with 𝑛.

To see this, suppose that 𝑝|𝑚. Then c≡me≡0 (mod p) and cd≡0≡m (mod p). We see that the (m^d)^e≡m property holds for all prime factors of n irrespective of whether or not they divide m and hence (m^d)^e mod n holds irrespective of whether or not m is coprime to n. For cryptographic n it is of course highly unlikely that m is not coprime to n.

Link to detailed proof: <https://crypto.stackexchange.com/questions/1004/does-rsa-work-for-any-message-m>

2.

In the RSA algorithm, it is better for the parameter *e* to be odd rather than even. This is because *e* must be coprime with *ϕ*(*n*), where *ϕ*(*n*)=(*p*−1)(*q*−1). Since *ϕ*(*n*) is always even (as both *p*−1 and *q*−1 are even), an even *e* would not be coprime with *ϕ*(*n*) unless it is 1, which is not suitable for encryption. Typically, small odd values like 3, 17, or 65537 are chosen for *e* as they are efficient for encryption and ensure the greatest common divisor with *ϕ*(*n*) is 1.

So while the RSA algorithm works mathematically for even values of e as well, choosing an odd public exponent e provides better security properties and computational advantages. Common values used for e in practice are 3, 17, and 65537 which are all odd.

3.

Fermat numbers are a special sequence of numbers defined by the formula:

*Fn*​=2^2^*n*+1

where *n* is a non-negative integer. The first few Fermat numbers are:

* *F*0​=3
* *F*1​=5
* *F*2​=17
* *F*3​=257
* *F*4​=65537

**Role in RSA Parameter Generation**

In the RSA algorithm, choosing the public exponent *e* as a Fermat number, particularly *F*4​=65537, is common. This choice is favored because:

1. **Efficiency:** Using 65537 (which is 2^16+1) as *e* ensures efficient encryption and signature verification due to its low Hamming weight (i.e., it has only two 1-bits in its binary representation).
2. **Security:** Fermat numbers, being relatively large primes, maintain the necessary property that *e* is coprime with *ϕ*(*n*).
3. **Necessity:** They are prime numbers, which is a requirement for the RSA algorithm.
4. **Odd numbers:** They are odd numbers and we rather e to be odd as stated in question 2.

So in summary, while not strictly required, using a Fermat prime number like 65537 as the public exponent e in RSA is a commonly recommended practice to generate strong and interoperable RSA parameters. 65537 strikes a balance between computational efficiency and security, making it a popular choice in RSA implementations.

4.

The RSA paper mentions a few efficient algorithms for performing the critical modular exponentiation operations M^e mod n and C^d mod n during encryption and decryption respectively. Some examples of optimal algorithms discussed are:

1. Exponentiation by repeated squaring and multiplication (Section VII.A)

This is a basic algorithm presented in the paper itself. It computes M^e mod n by repeatedly squaring M and multiplying by M when the corresponding bit in the binary representation of e is 1. Its time complexity is O(log e).

2. More efficient procedures (Section VII.A)

The paper mentions that more efficient procedures than the basic repeated squaring method are known, without going into details. Some examples are:

a) Sliding Window Exponentiation - Precomputes and reuses small powers to speed up exponentiation.

b) Addition Chain Exponentiation - Computes M^e by exploiting the addition chain for e.

3. Algorithms studied by Knuth (Section VII.A)

The paper cites Knuth's seminal book "The Art of Computer Programming" which studies exponentiation algorithms in great detail, including:

a) Binary exponentiation

b) Use of addition chains

c) Exploiting special exponent patterns

d) Trading multiplications for squarings

4. Montgomery multiplication algorithm

While not explicitly mentioned in the original RSA paper, the Montgomery multiplication method is an optimal technique to perform modular multiplications during the exponentiation, avoiding costly division operations.

**5. Barrett Reduction:**

This is used to improve the efficiency of the modular reduction step during exponentiation. It precomputes values that allow modular reduction to be performed without division, using multiplication and subtraction instead.

These and other advanced techniques like use of special modular arithmetic hardware can significantly optimize the core modular exponentiation operations in RSA, which are its most compute-intensive parts. Optimal exponentiation algorithms are critical for high performance RSA implementations.

5.

The security level of RSA key sizes is often compared to symmetric key sizes for block ciphers:

* A **1024-bit RSA** key is roughly equivalent to an **80-bit** symmetric key in terms of security.
* A **2048-bit RSA** key is roughly equivalent to a **112-bit** symmetric key in terms of security.

These comparisons provide an idea of how the computational effort required to break RSA encryption compares to that required to break symmetric encryption algorithms like AES.

However, it's important to note that these are just rough approximations. The computational complexity assumptions and attack models for public-key cryptosystems like RSA are quite different from those for block ciphers. But this estimation does give a general sense of the targeted security levels for different RSA key sizes compared to block ciphers.

6.

Generating proper prime numbers for RSA involves several steps to ensure the numbers are secure and suitable for cryptographic use:

**1. Random Selection of Candidates**

* **Bit Length:** Choose the bit length of the prime (e.g., 1024 bits for 2048-bit RSA keys).
* **Random Number Generation:** Generate a random number of the desired bit length. Ensure it's odd (as even numbers > 2 aren't prime).

**2. Primality Testing**

* **Basic Tests:** Perform initial checks like divisibility by small primes to quickly rule out non-primes.
* **Probabilistic Primality Tests:** Use tests such as Miller-Rabin or the Baillie-PSW test to determine if a number is prime with high probability. Repeat multiple rounds to reduce the chance of false positives.

**3. Ensuring Cryptographic Security**

* **Sufficient Rounds:** For Miller-Rabin, use enough rounds (e.g., 40 for high security) to achieve the desired level of confidence.
* **Avoiding Small Factors:** Ensure that the prime is not too close to powers of small primes to avoid certain cryptographic attacks.

**4. Verification and Validation**

* **Unique and Large Enough:** Ensure the primes *p* and *q* are distinct and large enough to provide the necessary security margin.
* **Additional Criteria:** Optionally, check for additional properties like *p*−1 or *q*−1 having large prime factors to enhance security.

**Example Process:**

1. **Generate Candidate:**
   * Generate a random 1024-bit odd number.
2. **Basic Check:**
   * Check if the number is divisible by any small prime (e.g., up to 1000).
3. **Primality Test:**
   * Apply the Miller-Rabin test for 40 iterations.
4. **Repeat:**
   * If the number passes all tests, it is likely prime; otherwise, generate a new candidate and repeat.

**Ensuring Safe Prime Generation:**

* Use cryptographically secure random number generators (CSPRNG).
* Validate implementation against known standards (e.g., FIPS 186-4).

**Libraries and Tools:**

* Cryptographic libraries like OpenSSL and GNU MP (GMP) provide built-in functions for generating and testing large primes, ensuring reliable and efficient prime number generation for RSA.

همچنین باید در نظر داشت که در حالت کلی الگوریتم های تولید اعداد اول به دو دسته تقسیم می شوند: الگوریتم های احتمالی و قطعی

الگوریتم های احتمالی تولید اعداد اول احتمال زیادی برای تولید یک عدد اول در یک محدوده مشخص ارائه می کنند، اما ممکن است گاهی اوقات اعداد ترکیبی تولید کنند. این الگوریتم ها اغلب برای آزمایش اولیه یا زمانی که سرعت بر قطعیت مطلق اولویت دارد استفاده میشود.

الگوریتم های رایج احتمالی تولید اعداد اول عبارتند از:

* Miller-Rabin
* Probabilistic Lucas
* ...

الگوریتم های تولید اعداد اول قطعی تضمین می کنند که خروجی یک عدد اول است، اما آنها ا**غلب کندتر** از الگوریتم های احتمالی هستند. این الگوریتم ها معمولا بر ویژگی های ریاضی خاص اعداد اول تکیه می کنند و ممکن است محاسبات پیچیدەتر و زمان طولانی تری را شامل شوند. الگوریتم های متداول تولید اعداد اول قطعی عبارتند از:

* Pollard’s rho algorithm
* Sieve of Eratosthenes