MA7010 – Number Theory for Cryptography - Assignment 1

Ajeesh Thattukunnel Vijayan

January 11th 2024

1 Notes

I have used a combination of Maple and Rust Code to arrive at the solutions. The code snippets presented in this document are in Rust. I developed the code using the u64 primitive datatype in Rust and later changed that to BigInt with the hope that I could use very large Integers such as more than 500bits long, but it became a challenge. Many times computer terminated the execution with Out Of Memory errors.

2 Answers

1. Lower Range = 2800, Upper Range = 3100.

   (a) List the elements of the set A = all primes p in the range, B = all composite numbers in the range.

   Answer:

$$A = [2801, 2803, 2819, 2833, 2837, 2843, 2851, 2857, 2861, 2879, 2887, 2897, 2903, 2909, 2917, 2927, 2939, 2953, 2957, 2963, 2969, 2971, 2999, 3001, 3011, 3019, 3023, 3037, 3041, 3049, 3061, 3067, 3079, 3083, 3089]$$

$$B = [2800, 2802, 2804, 2805, 2806, 2807, 2808, 2809, 2810, 2811, 2812, 2813, 2814, 2815, 2816, 2817, 2818, 2820, 2821, 2822, 2823, 2824, 2825, 2826, 2827, 2828, 2829, 2830, 2831, 2832, 2834, 2835, 2836, 2838, 2839, 2840, 2841, 2842, 2844, 2845, 2846, 2847, 2848, 2849, 2850, 2852, 2853, 2854, 2855, 2856, 2858, 2859, 2860, 2862, 2863, 2864, 2865, 2866, 2867, 2868, 2869, 2870, 2871, 2872, 2873, 2874, 2875, 2876, 2877, 2878, 2880, 2881, 2882, 2883, 2884, 2885, 2886, 2888, 2889, 2890, 2891, 2892, 2893, 2894, 2895, 2896, 2898, 2899, 2900, 2901, 2902, 2904, 2905, 2906, 2907, 2908, 2910, 2911, 2912, 2913, 2914, 2915, 2916, 2918, 2919, 2920, 2921, 2922, 2923, 2924, 2925, 2926, 2928, 2929, 2930, 2931, 2932, 2933, 2934, 2935, 2936, 2937, 2938, 2940, 2941, 2942, 2943, 2944, 2945, 2946, 2947, 2948, 2949, 2950, 2951, 2952, 2954, 2955, 2956, 2958, 2959, 2960, 2961, 2962, 2964, 2965, 2966, 2967, 2968, 2970, 2972, 2973, 2974, 2975, 2976, 2977, 2978, 2979, 2980, 2981, 2982, 2983, 2984, 2985, 2986, 2987, 2988, 2989, 2990, 2991, 2992, 2993, 2994, 2995, 2996, 2997, 2998, 3000, 3002, 3003, 3004, 3005, 3006, 3007, 3008, 3009, 3010, 3012, 3013, 3014, 3015, 3016, 3017, 3018, 3020, 3021, 3022, 3024, 3025, 3026, 3027, 3028, 3029, 3030, 3031, 3032, 3033, 3034, 3035, 3036, 3038, 3039, 3040, 3042, 3043, 3044, 3045, 3046, 3047, 3048, 3050, 3051, 3052, 3053, 3054, 3055, 3056, 3057, 3058, 3059, 3060, 3062, 3063, 3064, 3065, 3066, 3068, 3069, 3070, 3071, 3072, 3073, 3074, 3075, 3076, 3077, 3078, 3080, 3081, 3082, 3084, 3085, 3086, 3087, 3088, 3090, 3091, 3092, 3093, 3094, 3095, 3096, 3097, 3098, 3099, 3100]$$

The below images depicts the execution of the code on a powershell terminal:

```
PS D:\workspace\nt-assignments> .\target\debug\nt-assignments.exe list-primes -s 2800 -e 3100
```

Prime Numbers:

Number	2851	2909	2969	3037	3089
2801	2857	2917	2971	3041	
2803	2861	2927	2999	3049	
2819	2879	2939	3001	3061	
2833	2887	2953	3011	3067	
2837	2897	2957	3019	3079	
2843	2903	2963	3023	3083	

Figure 1: Prime Numbers - Code Execution Output

Composite Numbers:

Number	2821	2842	2865	2885	2907	2929	2949	2973	2992	3014	3035	3057	3078	3100
2800	2822	2844	2866	2886	2908	2930	2950	2974	2993	3015	3036	3058	3080	
2802	2823	2845	2867	2888	2910	2931	2951	2975	2994	3016	3038	3059	3081	
2804	2824	2846	2868	2889	2911	2932	2952	2976	2995	3017	3039	3060	3082	
2805	2825	2847	2869	2890	2912	2933	2954	2977	2996	3018	3040	3062	3084	
2806	2826	2848	2870	2891	2913	2934	2955	2978	2997	3020	3042	3063	3085	
2807	2827	2849	2871	2892	2914	2935	2956	2979	2998	3021	3043	3064	3086	
2808	2828	2850	2872	2893	2915	2936	2958	2980	3000	3022	3044	3065	3087	
2809	2829	2852	2873	2894	2916	2937	2959	2981	3002	3024	3045	3066	3088	
2810	2830	2853	2874	2895	2918	2938	2960	2982	3003	3025	3046	3068	3090	
2811	2831	2854	2875	2896	2919	2940	2961	2983	3004	3026	3047	3069	3091	
2812	2832	2855	2876	2898	2920	2941	2962	2984	3005	3027	3048	3070	3092	
2813	2834	2856	2877	2899	2921	2942	2964	2985	3006	3028	3050	3071	3093	
2814	2835	2858	2878	2900	2922	2943	2965	2986	3007	3029	3051	3072	3094	
2815	2836	2859	2880	2901	2923	2944	2966	2987	3008	3030	3052	3073	3095	
2816	2838	2860	2881	2902	2924	2945	2967	2988	3009	3031	3053	3074	3096	
2817	2839	2862	2882	2904	2925	2946	2968	2989	3010	3032	3054	3075	3097	
2818	2840	2863	2883	2905	2926	2947	2970	2990	3012	3033	3055	3076	3098	
2820	2841	2864	2884	2906	2928	2948	2972	2991	3013	3034	3056	3077	3099	

Figure 2: Composite Numbers - Code Execution Output

Code Snippet - Prime Number Sieve

```

    /// Returns a boolean representing if the given number is prime or
 not
    ///
    /// # Arguments
    ///
    /// * ‘n‘ - A BigInt
    ///
    /// # Examples
    ///
    /// ‘‘‘
    /// use crate::primality::is_prime_trial_division_parallel;
```

```rust
    /// let is_prime = is_prime_trial_division_parallel(BigInt::from(100u64));
    /// ```
    pub fn is_prime_trial_division_parallel(n: &BigInt) -> bool {
      let (zero, one, _two) = (BigInt::from(0u64), BigInt::from(1u64), BigInt::from(2u64));
      let three = BigInt::from(3u64);

      // returns true if the number is 2 or 3
      if n <= &three {
        return n > &one;
      }

      if n % 2 == zero || n % 3 == zero {
        return false;
      }

      let upper_bound = n.sqrt() + 1; // +1 to get the ceiling value

      if let Some(_divisor) = range_inclusive(BigInt::from(5u64), upper_bound)
      .par_bridge()
      .into_par_iter()
      .find_first(|divisor| n % divisor == zero)
      {
        false
      } else {
        true
      }
    }
```

Listing 1: Prime Number Sieve

The above code verifies the primality of a number using trial division. It generates a sequence of numbers from 2 to $sqrt(n) + 1$ and divides these numbers into chunks of blocks and checks the divisibility in parallel to speed up the execution. The parallelisation library used for this purpose is Rayon

The below command execute the Prime Number Sieve:

```
.\nt-assignments.exe list-primes -s 2800 -e 3100
```

Listing 2: Example command - Prime Number Sieve

(b) List the elements of the set C where C = {composite numbers n = pq in your range which are the product of exactly two distinct primes p and q}.

Answer: The code snippet below extracts the numbers of the form $n = p.q$

```rust
    ///
    /// Returns a tuple with a formatted string for output and a Vector which contains a tuple of
    /// Number and its prime factors
    ///
    /// # Arguments
    /// * `start` - BigInt
```

````rust
    /// * `end` - BigInt
    /// * `NumCategory` - Whether we want the prime factorisation of All numbers or composites or composits of the form P.Q
    /// # Example
    /// ```
    /// use crate::presets::list_prime_factors_in_range;
    /// list_prime_factors_in_range(&start, &end, NumCategory::All);
    /// ```
    pub fn list_prime_factors_in_range(
    start: &BigInt,
    end: &BigInt,
    opts: NumCategory,
    ) -> (Vec<NumFactorTable>, Vec<(BigInt, Vec<(BigInt, usize)>)>) {
      let mut table_data: Vec<NumFactorTable> = Vec::new();
      let mut primes = vec![BigInt::from(2u64)];
      let mut nums_pfactors: Vec<(BigInt, Vec<(BigInt, usize)>)> = Vec::new();
      for num in range_inclusive(start.clone(), end.clone()) {
        let mut form: String = String::new();
        let p_factors = num.prime_factors(&mut primes);
        match opts {
          NumCategory::All => {
            format_prime_factors_print(&num, &p_factors, &mut form, &mut table_data);
            nums_pfactors.push((num.clone(), p_factors.clone()));
          }
          NumCategory::Composites => {
            if p_factors.len() >= 2 {
              format_prime_factors_print(&num, &p_factors, &mut form, &mut table_data);
              nums_pfactors.push((num.clone(), p_factors.clone()));
            }
          }
          NumCategory::CompositesPQ => {
            if p_factors.len() == 2 {
              let first = p_factors.first().unwrap();
              let second = p_factors.get(1).unwrap();

              match first.1 {
                1 => match second.1 {
                  1 => {
                    format_prime_factors_print(
                    &num,
                    &p_factors,
                    &mut form,
                    &mut table_data,
                    );
                    nums_pfactors.push((num.clone(), p_factors.clone()));
                  }
                  _ => {}
                },
                _ => {}
              }
            }
          }
          NumCategory::Primes => {}
        }
      }

      (table_data, nums_pfactors)
    }
````

```

    pub trait PrimeFactors {
      fn prime_factors(&self, primes: &mut Vec<BigInt>) -> Vec<(BigInt, usize)>;
      //fn is_prime_factors_form_pq(&self) -> (bool, Vec<(BigInt, usize)>);
    }

    impl PrimeFactors for BigInt {
      fn prime_factors(&self, primes: &mut Vec<BigInt>) -> Vec<(Self, usize)> {
        let n = self.clone();
        // Check if n is prime
        if miller_rabin_primality(&self) {
          return vec![(self.clone(), 1)];
        }

        let start_no = primes.last().unwrap();
        let square_root = self.sqrt();
        if square_root - start_no > BigInt::from(2u64) {
          let end_no: BigInt = self.sqrt() + 1; // +1 to get the ceiling value
          // println!("start = {}, end = {}", start_no, end_no);

          let r = range_inclusive(start_no.clone(), end_no);

          let new_primes: Vec<BigInt> = r
          .into_iter()
          .map(|x| x)
          .parallel_filter(|x| miller_rabin_primality(x))
          .collect();
          primes.extend(new_primes);
          let mut seen = HashSet::new();
          primes.retain(|c| seen.insert(c.clone()));
        }
        let _res: HashMap<BigInt, usize> = HashMap::new();

        // The all_divisors vec will contain all the divisors of num with repetition.
        // The product of the elements of all_divisors will equal the "num"
        let mut all_divisors = Vec::<BigInt>::new(); //
        let mut product = BigInt::one();

        while product < n {
          let divisors = primes
          .par_iter()
          .filter(|x| (n.clone() / &product) % *x == BigInt::zero())
          .map(|p| p.clone())
          .collect::<Vec<BigInt>>();
          all_divisors.extend(divisors.clone());
          product = product
          * divisors
          .iter()
          .fold(BigInt::one(), |acc: BigInt, a| acc * a);
          let q = &n / &product;
          if miller_rabin_primality(&q) {
            all_divisors.push(q);
            break;
          }
        }

```

```
            let mut res = all_divisors
            .into_iter()
            .fold(HashMap::<BigInt, usize>::new(), |mut m, x| {
              *m.entry(x).or_default() += 1;
              m
            })
            .into_iter()
            .filter_map(|(k, v)| Some((k, v)))
            .collect::<Vec<(BigInt, usize)>>();
            res.sort_by_key(|k| k.0.clone());
            res
        }
      }

```

Listing 3: Prime Factorisation

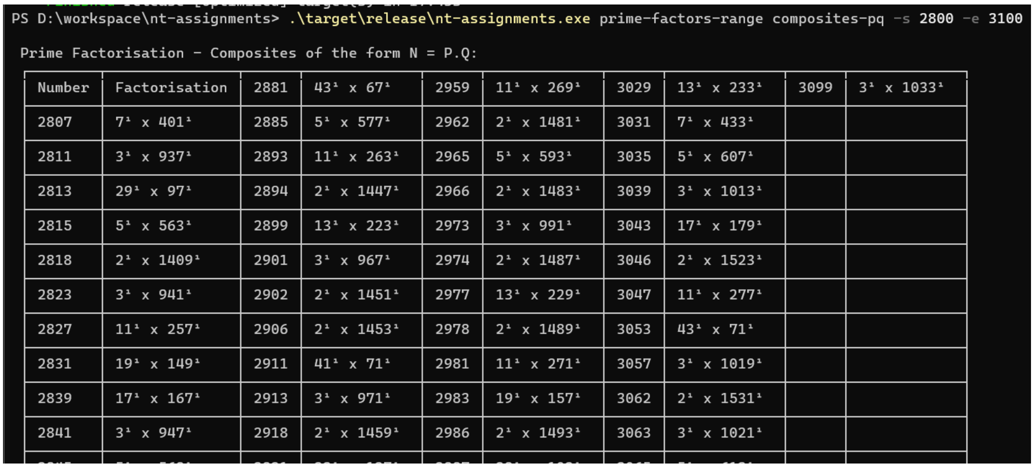
The above two Rust procedures handle the prime factorisation of the integers in the given range. The below snippet extract the numbers of the form $p.q$

```

      NumCategory::CompositesPQ => {
        if p_factors.len() == 2 {
          let first = p_factors.first().unwrap();
          let second = p_factors.get(1).unwrap();

          match first.1 {
            1 => match second.1 {
              1 => {
                format_prime_factors_print(
                &num,
                &p_factors,
                &mut form,
                &mut table_data,
                );
                nums_pfactors.push((num.clone(), p_factors.clone()));
              }
              _ => {}
            },
            _ => {}
          }
        }
      }

```

Listing 4: Code - Prime Factorisation - Search for ‘p.q‘”

Composites of the form $N = P.Q$					
Number	Factorisation	Number	Factorisation	Number	Factorisation
2807	$7^1 \times 401^1$	2811	$3^1 \times 937^1$	2813	$29^1 \times 97^1$
2815	$5^1 \times 563^1$	2818	$2^1 \times 1409^1$	2823	$3^1 \times 941^1$
2827	$11^1 \times 257^1$	2831	$19^1 \times 149^1$	2839	$17^1 \times 167^1$
2841	$3^1 \times 947^1$	2845	$5^1 \times 569^1$	2846	$2^1 \times 1423^1$
2854	$2^1 \times 1427^1$	2855	$5^1 \times 571^1$	2858	$2^1 \times 1429^1$
2859	$3^1 \times 953^1$	2863	$7^1 \times 409^1$	2866	$2^1 \times 1433^1$
2867	$47^1 \times 61^1$	2869	$19^1 \times 151^1$	2878	$2^1 \times 1439^1$
2881	$43^1 \times 67^1$	2885	$5^1 \times 577^1$	2893	$11^1 \times 263^1$
2894	$2^1 \times 1447^1$	2899	$13^1 \times 223^1$	2901	$3^1 \times 967^1$
2902	$2^1 \times 1451^1$	2906	$2^1 \times 1453^1$	2911	$41^1 \times 71^1$
2913	$3^1 \times 971^1$	2918	$2^1 \times 1459^1$	2921	$23^1 \times 127^1$
2923	$37^1 \times 79^1$	2929	$29^1 \times 101^1$	2931	$3^1 \times 977^1$
2933	$7^1 \times 419^1$	2935	$5^1 \times 587^1$	2941	$17^1 \times 173^1$
2942	$2^1 \times 1471^1$	2947	$7^1 \times 421^1$	2949	$3^1 \times 983^1$
2951	$13^1 \times 227^1$	2959	$11^1 \times 269^1$	2962	$2^1 \times 1481^1$
2965	$5^1 \times 593^1$	2966	$2^1 \times 1483^1$	2973	$3^1 \times 991^1$
2974	$2^1 \times 1487^1$	2977	$13^1 \times 229^1$	2978	$2^1 \times 1489^1$
2981	$11^1 \times 271^1$	2983	$19^1 \times 157^1$	2986	$2^1 \times 1493^1$
2987	$29^1 \times 103^1$	2991	$3^1 \times 997^1$	2993	$41^1 \times 73^1$
2995	$5^1 \times 599^1$	2998	$2^1 \times 1499^1$	3005	$5^1 \times 601^1$
3007	$31^1 \times 97^1$	3013	$23^1 \times 131^1$	3017	$7^1 \times 431^1$
3022	$2^1 \times 1511^1$	3027	$3^1 \times 1009^1$	3029	$13^1 \times 233^1$
3031	$7^1 \times 433^1$	3035	$5^1 \times 607^1$	3039	$3^1 \times 1013^1$
3043	$17^1 \times 179^1$	3046	$2^1 \times 1523^1$	3047	$11^1 \times 277^1$
3053	$43^1 \times 71^1$	3057	$3^1 \times 1019^1$	3062	$2^1 \times 1531^1$
3063	$3^1 \times 1021^1$	3065	$5^1 \times 613^1$	3071	$37^1 \times 83^1$
3073	$7^1 \times 439^1$	3077	$17^1 \times 181^1$	3085	$5^1 \times 617^1$
3086	$2^1 \times 1543^1$	3091	$11^1 \times 281^1$	3093	$3^1 \times 1031^1$
3095	$5^1 \times 619^1$	3097	$19^1 \times 163^1$	3098	$2^1 \times 1549^1$
3099	$3^1 \times 1033^1$	-	-	-	-

Table 1: List of composite numbers of the form P.Q



PS D:\workspace\nt-assignments> .\target\release\nt-assignments.exe prime-factors-range composites-pq -s 2800 -e 3100

Prime Factorisation - Composites of the form N = P.Q:

Number	Factorisation	2881	43^1 x 67^1	2959	11^1 x 269^1	3029	13^1 x 233^1	3099	3^1 x 1033^1
2807	7^1 x 401^1	2885	5^1 x 577^1	2962	2^1 x 1481^1	3031	7^1 x 433^1		
2811	3^1 x 937^1	2893	11^1 x 263^1	2965	5^1 x 593^1	3035	5^1 x 607^1		
2813	29^1 x 97^1	2894	2^1 x 1447^1	2966	2^1 x 1483^1	3039	3^1 x 1013^1		
2815	5^1 x 563^1	2899	13^1 x 223^1	2973	3^1 x 991^1	3043	17^1 x 179^1		
2818	2^1 x 1409^1	2901	3^1 x 967^1	2974	2^1 x 1487^1	3046	2^1 x 1523^1		
2823	3^1 x 941^1	2902	2^1 x 1451^1	2977	13^1 x 229^1	3047	11^1 x 277^1		
2827	11^1 x 257^1	2906	2^1 x 1453^1	2978	2^1 x 1489^1	3053	43^1 x 71^1		
2831	19^1 x 149^1	2911	41^1 x 71^1	2981	11^1 x 271^1	3057	3^1 x 1019^1		
2839	17^1 x 167^1	2913	3^1 x 971^1	2983	19^1 x 157^1	3062	2^1 x 1531^1		
2841	3^1 x 947^1	2918	2^1 x 1459^1	2986	2^1 x 1493^1	3063	3^1 x 1021^1		

Figure 3: Sample output on a Windows terminal

The below command execution prints numbers of the form $n = p.q$ in a table:

```
.\target\release\nt-assignments.exe prime-factors-range composites-pq -s 2800 -e 3100
```

Listing 5: Print numbers of the form n

(c) Choose any three element of the set B and then randomly select 4 values of a for each element.

Apply the gcd test for each of the 12 cases and report on how accurate it is in determining that a number is composite.

Answer: The below image shows the output of one execution of the gcd test on three composite numbers selected random in the inclusive range of 2800 to 3100.

```
PS D:\workspace\nt-assignments> .\target\release\nt-assignments.exe primality gcd -s 2800 -e 3100
```

n = p.q	a (randomly selected)	gcd(n, a)
2914 = 2^1 x 31^1 x 47^1	a1 = 517	gcd1 = 47
	a2 = 1710	gcd2 = 2
	a3 = 458	gcd3 = 2
	a4 = 1341	gcd4 = 1
2877 = 3^1 x 7^1 x 137^1	a1 = 1732	gcd1 = 1
	a2 = 1799	gcd2 = 7
	a3 = 2525	gcd3 = 1
	a4 = 338	gcd4 = 1
2895 = 3^1 x 5^1 x 193^1	a1 = 1421	gcd1 = 1
	a2 = 1618	gcd2 = 1
	a3 = 1891	gcd3 = 1
	a4 = 1883	gcd4 = 1

Figure 4: Primality Check using GCD Test

At the first glance we could see that the composite number $n = 2895$ which has a prime factorisation of $3^1 \times 5^1 \times 193^1$ do not have any Fermat Witnesses to prove that it's a composite number. All the randomly selected $a = \{1421, 1618, 1891, 1883\}$ values yielded $gcd = 1$ which makes all these a values Fermat Liars.

The accuracy of GCD Test for primality depends on the selection of the a values. Of course it's not practical to test with all the numbers less than n to find if n is composite. It will turn into the sieving process if we do that. Also, there are cases where some numbers (Carmichael Numbers) do not yield any Fermat Witnesses. The Euler Totient Function $\phi(n)$ gives the total number of relatively prime numbers less than n. Which means for a composite number n, $n - \phi(n)$ values will attest n is composite. $n - \phi(n)$ becomes smaller when $\phi(n)$ is large. For composite numbers of the form $n = p.q$, that's numbers with fewer prime factors have higher values of $\phi(n)$.

Let's consider the number $n = 2881$

$$
\begin{aligned}
2881 &= 43^1 \times 67^1 && \text{(prime factorisation)}\\
\phi(2881) &= 42 \times 66 = 2772\\
n - \phi(n) &= 109 \approx 4\%
\end{aligned}
$$

Only 4% of the numbers are Fermat Witnesses in this case which is much much smaller to form an definite opinion on whether such a number is prime or not when we choose the bases randomly.

The below command execution prints output of GCD Test in a table:

```

.\target\release\nt-assignments.exe primality gcd -s 2800 -e 3100

```

Listing 6: GCD Test Execution

GCD Test Code snippet:

````rust

/// Returns a Vec of randomly selected `a` value and `gcd`
///
/// # Arguments
/// * n - BigInt - Number for which we are checking primality
/// * num_trials - u8 - How many trials we do
///
/// # Examples
/// ```
/// use crate::primality::gcd_test
/// let result: Vec<(BigInt, BigInt)> = gcd_test(&BigInt::from(2881u64), 4);
/// ```
///
pub fn gcd_test(n: &BigInt, num_trials: u8) -> Vec<(BigInt, BigInt)> {
  let mut r = Vec::<BigInt>::new();
  for _ in 0..num_trials {
    r.push(generate_random_int_in_range(&BigInt::from(2u8), &(n - 1)));
  }

  let mut result = Vec::<(BigInt, BigInt)>::new();
  for a in r.iter() {
    result.push((a.clone(), n.gcd_euclid(&a)));
  }

  result
}

pub trait Gcd {
  ///
  /// # Examples
  ///
  /// ```
  /// use utils::Gcd;
  ///
  /// assert_eq!(BigInt::from(44u64), BigInt::from(2024u64).gcd_euclid(&BigInt::from(748u64)));
  /// ```

  /// Determine [greatest common divisor](https://en.wikipedia.org/wiki/Greatest_common_divisor)
  /// using the [Euclidean algorithm](https://en.wikipedia.org/wiki/Euclidean_algorithm).
  fn gcd_euclid(&self, other: &Self) -> Self;
}

impl Gcd for BigInt {
  ///
  /// GCD Calculator - The Euclidean Algorithm
  /// Input: A pair of integers a and b, not both equal to zero
  /// Output: gcd(a, b)
  ///
  fn gcd_euclid(&self, other: &BigInt) -> BigInt {
    let zero = BigInt::from(0u64);
    let mut a = self.clone();
    let mut b = other.clone();
    let mut gcd: BigInt = zero.clone();
    if b > a {
      gcd = b.gcd_euclid(&a);
````

```
        } else {
          let mut r: BigInt = &a % &b;
          while &r > &zero {
            // let q = &a / &b;
            r = &a % &b;

            if &r != &zero {
              a = b;
              b = r.clone();
            }
          }

          gcd = b;
        }

        gcd
      }
    }

```

Listing 7: Code - Primality using GCD Test"

2. Find all Carmichael Numbers in your range (Lower Range = 2800, Upper Range = 3100) using:

   (a) A direct method employing the Fermat Test that shows that a composite number n has no Fermat Witnesses.

   Answer: The below code snippet shows how FLT is employed in finding a Carmichael number:

```

    /// Returns a list of Carmichael Numbers (Absolute Pseudoprimes) in a range using FLT or Korselt's criterion
    ///
    /// # Arguments
    /// * start: BigInt
    /// * end: BigInt
    /// * f: a function pointer to either primality::carmichael_nums_korselt or primality::carmichael_nums_flt
    /// # Examples
    /// ```
    /// use crate::presets::list_carmichael_nums;
    /// let carmichael_nums = list_carmichael_nums(&start, &end, carmichael_nums_flt);
    /// ```
    ///
    pub fn list_carmichael_nums(start: &BigInt, end: &BigInt, f: fn(&BigInt) -> bool) -> (String, Vec<(BigInt, Vec<(BigInt, usize)>)>) {
      // Get all the composite numbers in the range
      let composites = list_prime_factors_in_range(start, end, NumCategory::Composites).1;

      // Searching for Carmichael numbers in parallel
      let carmichael_nums = composites
      .par_iter()
      .filter(|x| f(&x.0) == true)
      .map(|x| x.clone())
      .collect::<Vec<(BigInt, Vec<(BigInt, usize)>)>>();

      // Format the data for printing
```

```
        let mut table_data: Vec<NumFactorTable> = Vec::new();
        for item in carmichael_nums.iter() {
          let mut form: String = String::new();
          format_prime_factors_print(&item.0, &item.1, &mut form, &mut
    table_data);
        }

        let mut table1 = Table::new(table_data);
        table1.with(STYLE_2);

        let output1 = table1.to_string();
        (output1, carmichael_nums)
      }

      ///
      /// Carmichael Numbers using FLT
      /// n: a composite number
      ///
      pub fn carmichael_nums_flt(n: &BigInt) -> bool {
        let n_minus_one = n - 1;
        // Get all the coprime numbers less than 'n'
        let coprimes_n = coprime_nums_less_than_n(n);

        // Search for Fermat Witnesses. A Fermat Witness will yeild
    a^{n-1} ≢ 1(mod n)
        let fermat_witnesses = coprimes_n
        .par_iter()
        .filter(|x| modular_pow(&x, &n_minus_one, n) != BigInt::one())
        .map(|x| x.clone())
        .collect::<Vec<BigInt>>();

        // No Fermat Witness means n is a Carmichael Number
        fermat_witnesses.len() == 0
      }

```

Listing 8: Code - Search Carmichael Numbers in the range"

When we run the above code, we get $2821 = 7^1 \times 13^1 \times 31^1$ as the Carmichael Number between 2800 and 3100 inclusive. A sample execution is given below:

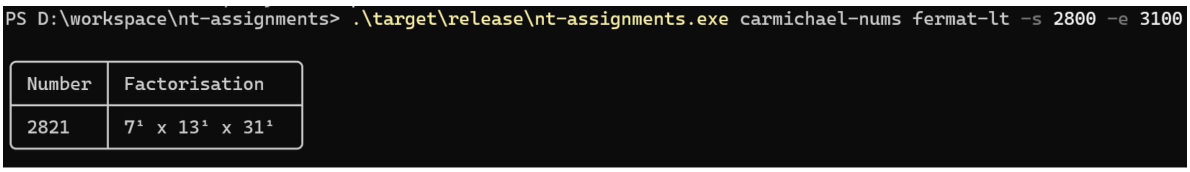


Figure 5: Carmichael Number using FLT- Example result

The below command execution prints Carmichael Numbers in the range using FLT:

```

    .\target\release\nt-assignments.exe carmichael-nums fermat-lt -s 2800 -e 3100

```

Listing 9: Carmichael Numbers using FLT

(b) Checking which numbers satisfy Korselt's Criteria.

Answer: Korselt's criteria states:

1. n is squarefree i.e. the prime decomposition of n do not contain any repeated factors;
2. $p|n \implies (p-1)|(n-1)$;

The below code snippet is the implementation of the above criteria:

```
    ///
    /// Carmichael Numbers using Korselt's criteria
    /// n: a composite number
    ///
    pub fn carmichael_nums_korselt(n: &BigInt) -> bool {
      // initialisation to search prime factors
      let mut primes = vec![BigInt::from(2u64)];
      // prime factorisation of `n`
      let p_factors = n.prime_factors(&mut primes);
      // checking if the number is squarefree
      let squarefree = p_factors.iter().fold(true, |squarefree: bool, factor| {
        squarefree & (factor.1 == 1)
      });

      let mut p_m_o_divides_n_m_o = true;
      // if the number is squarefree, then check if `p minus one` divides `n minus one`
      if squarefree {
        let n_minus_one = n - 1;
        for (p, _) in p_factors.iter() {
          p_m_o_divides_n_m_o &= &n_minus_one % (p - 1) == BigInt::zero();
        }
      }

      // if both are true, return true
      squarefree & p_m_o_divides_n_m_o
    }
```

Listing 10: Carmichael Number Check - Korselt's criteria

3. Take the first five elements n of the set B of composite numbers with 2 factors in your range (or all numbers if you find there are less than 10). The Miller Rabin test states that at most $\frac{1}{4}$ of numbers a that are randomly chosen will give the answer that n is 'probably prime'. How close can you get to this maximum, (i.e. which of your 5 choices has the highest proportion of possible a's that would fail the Miller Rabin test).

   What composite numbers m between 50 and 100 have the highest proportion of Miller Rabin failures? (For each number in the range work out the proportion of a's that produce the answer 'm is probably prime'). Look at the prime factorisation of these numbers and see if it suggests any patterns about which numbers are vulnerable to giving false answers in Miller Rabin.

   Answer: For this question, I have filtered out the odd numbers with 2 factors from the set B of composite numbers. There were 82 such numbers. When I looked for the Miller-Rabin non-witnesses for the first 5 elements, only one number had non-witnesses. Hence I have considered the whole set of odd composites with two factors, i.e., all the 82 numbers and 31 numbers have non-witnesses. The numbers with liars are listed below:

   Numbers with Miller-Rabin Liars in the range $2800 \leq n \leq 3100$:

$$\begin{aligned}&\{2813, 2825, 2845, 2863, 2869, 2873, 2881, 2885, 2899, 2911,\\&2923, 2929, 2941, 2947, 2965, 2977, 2981, 2983, 2989, 2993,\\&3005, 3007, 3029, 3031, 3053, 3065, 3073, 3077, 3085, 3091, 3097\}\end{aligned} \tag{1}$$

For our study, we will consider the first 5 numbers from the above set. Let N be that set.
Let $N = \{2813, 2825, 2845, 2863, 2869\}$

1. $n = 2813$

$$
\begin{aligned}
&2813 = 29^1 \times 97^1 && \text{(prime factorisation)}\\
&n-1 = 2812 = 703.2^2 && (n-1 = m.2^s \text{ form, where } m = 703, s = 2)\\
&A = \{75, 1380, 1433, 2738\} && \text{(Set } A \text{ represents the bases that became liars)}\\
&|A| = 4
\end{aligned}
$$

The Miller–Rabin sequence for n generated by the set A is $(a_i^m, a_i^{2m}) \mod n$.

$$(75^{703}, 75^{2.703}) \mod 2813 = (2738, 2812) \tag{2}$$
$$(1380^{703}, 1380^{2.703}) \mod 2813 = (1433, 2812) \tag{3}$$
$$(1433^{703}, 1433^{2.703}) \mod 2813 = (1380, 2812) \tag{4}$$
$$(2738^{703}, 2738^{2.703}) \mod 2813 = (75, 2812) \tag{5}$$

For all the bases, the second number, $2812 \equiv -1 \mod 2813$ and hence 2813 is a prime with respect to these bases. In other words, $A = \{75, 1380, 1433, 2738\}$ are Miller-Rabin Liars for 2813. Similarly for the other bases.

2. $n = 2825$

$$
\begin{aligned}
&2825 = 5^2 \times 113^1 && \text{(prime factorisation)}\\
&n-1 = 2824 = 353.2^3 && (n-1 = m.2^s \text{ form, where } m = 353, s = 3)\\
&A = \{693, 1032, 1793, 2132\} && \text{(Set } A \text{ represents the bases that became liars)}\\
&|A| = 4
\end{aligned}
$$

3. $n = 2845$

$$
\begin{aligned}
&2845 = 5^1 \times 569^1 && \text{(prime factorisation)}\\
&n-1 = 2844 = 711.2^2 && (n-1 = m.2^s \text{ form, where } m = 711, s = 2)\\
&A = \{483, 1052, 1793, 2362\} && \text{(Set } A \text{ represents the bases that became liars)}\\
&|A| = 4
\end{aligned}
$$

4. $n = 2863$

$$
\begin{aligned}
&2863 = 7^1 \times 409^1 && \text{(prime factorisation)}\\
&n-1 = 2862 = 1431.2^1 && (n-1 = m.2^s \text{ form, where } m = 1431, s = 1)\\
&A = \{53, 54, 356, 410, 764, 817, 1173, 1174, 1689, 1690,\\
&\quad 2046, 2099, 2453, 2507, 2809, 2810\}\\
&|A| = 16
\end{aligned}
$$

5. $n = 2869$

$$
\begin{aligned}
&2869 = 19^1 \times 151^1 && \text{(prime factor}\\
&n-1 = 2868 = 717.2^2 && (n-1 = m.2\\
&A = \{334, 335, 571, 905, 938, 939, 1025, 1360, 1509, 1844, 1930, 1931, 1964, 2298, 2534, 2535\}\\
&|A| = 16
\end{aligned}
$$

From the set $N = \{2813, 2825, 2845, 2863, 2869\}$, we can see that the numbers 2863 and 2869 have 16 Miller-Rabin Liars each. Our bases selection is from the range $1 < a < n-1$, and the number of elements in this range coprime to n are $\phi(n)$. Only these coprime bases may report a number as pseudoprime. Hence we can see that by selecting the number 2863, we get the highest proportion $\frac{16}{\phi(2863)} = \frac{16}{2448} = \frac{1}{153}$ that the test falsely reporting a number as prime.

The below json listing presents all the numbers between 50 to 100 which are falsely identified as primes by the Miller-Rabin test against some of the bases used.

```
{
  "65": {
    "n - 1": 64 = 1.2^6,
    "prime factorisation": 5^1 × 13^1,
    "Nonwitnesses(Liars)": [ 8, 18, 47, 57 ]
  },
  "85": {
    "n - 1": 84 = 21.2^2,
    "prime factorisation": 5^1 × 17^1,
    "Nonwitnesses(Liars)": [ 13, 38, 47, 72 ]
  },
  "91": {
    "n - 1": 90 = 45.2^1,
    "prime factorisation": 7^1 × 13^1,
    "Nonwitnesses(Liars)": [ 9, 10, 12, 16, 17, 22, 29, 38,
      53, 62, 69, 74, 75, 79, 81, 82
    ]
  }
}

```

Listing 11: Miller-Rabin failues for numbers between 50 to 100

Let's calculate the proportion of a's that contribute to the false reporting are calculated below for each number:

1. $n = 65$

$$65 = 5^1 \times 13^1 \qquad \text{(prime factorisation)}$$
$$\phi(65) = 4 \times 12 = 48$$
$$\text{Number of MR Liars} = 4$$
$$\text{Hence the proportion coprimes which wrongly declares 65 as prime} = \frac{4}{48} = \frac{1}{12}$$

2. $n = 85$

$$85 = 5^1 \times 17^1 \qquad \text{(prime factorisation)}$$
$$\phi(65) = 4 \times 12 = 64$$
$$\text{Number of MR Liars} = 4$$
$$\text{Hence the proportion coprimes which wrongly declares 85 as prime} = \frac{4}{64} = \frac{1}{16}$$

3. $n = 91$

$$91 = 7^1 \times 13^1 \qquad \text{(prime factorisation)}$$
$$\phi(91) = 6 \times 12 = 72$$
$$\text{Number of MR Liars} = 16$$
$$\text{Hence the proportion coprimes which wrongly declares 85 as prime} = \frac{16}{72} = \frac{9}{12}$$

Observation: MR-Liars exist mostly for those numbers with distinct primes in its prime decomposition and many times the factors are squarefree. If there are only two prime factors in the prime decomposition of a number, and if the factors are of the form $p \equiv 1 \mod 4$, then there are 4 MR-Liars and if any of the factors are of the form $p \equiv 3 \mod 4$, then there are 16 or more MR-Liars exist. When there are three distinct prime factors, and two of them are $p \equiv 3 \mod 4$, then there are 8 or more MR-Liars exist.

Below listing shows the code used in finding the MR Liars for the numbers in the range $2800 \leq n \leq 3100$

```

    ///
    /// Miller-Rabin Test - Returns whether a number is prime or not
    ///
    /// # Arguments
    /// * n: BigInt
    /// * base: Optional - if base is not passed, `a` is randomly generated in the range
    ///         2 <= a <= n-2
    ///
    pub fn miller_rabin_test(n: &BigInt, base: Option<&BigInt>) -> (bool, Vec<MillerRabinTable>) {
      let mut table_data: Vec<MillerRabinTable> = Vec::new();
      let _is_prime = false;
      let (zero, one, two) = (BigInt::from(0u64), BigInt::from(1u64), BigInt::from(2u64));
      let n_minus_one: BigInt = n - 1;
      let mut m = n_minus_one.clone();

      let mut s = 0;
      while &m % 2 == zero {
        m /= 2;
        s += 1;
      }

      let n_minus_one_form = format!("{} = {}.2{}", n_minus_one, m, Superscript(s),);

      let a: BigInt;
      // If `base` is not passed, then randomly generate a base "a" such that 1 < a < n - 1
      if let Some(base) = base {
        a = base.clone();
      } else {
        a = generate_random_int_in_range(&two, &(n - 1));
      }
      // let a = BigInt::from(1003u64);

      // Calculate x ≡ a^m ( mod n)
      let mut x = modular_pow(&a, &m, n);

      format_miller_rabin_steps_print(
      n.clone(),
      &n_minus_one_form,
      s,
      a.clone(),
      0,
      m.clone(),
      x.clone(),
      &x == &one,
      &x == &(n - 1),
      &mut table_data,
      );

```

```
        // if $x \equiv 1 (\mod n)$,
        // We know that $a^{n-1} \equiv (a^{m.2^s}) \equiv 1 (\mod n)$, and we will not
        // find a square root of 1, other than 1, in repeated squaring
        // of $a^m$ to get $a^{n-1}$.
        if &x == &one || &x == &(n - 1) {
          return (true, table_data);
        }

        let mut k = 1;
        while k <= s - 1 {
          // searching square-roots for $1 (\mod n)$ other than $1 (\mod n)$
          let e = &m * BigInt::from(2u64).pow(k);
          x = modular_pow(&a, &e, n);

          format_miller_rabin_steps_print(
          n.clone(),
          &n_minus_one_form,
          s,
          a.clone(),
          k,
          e.clone(),
          x.clone(),
          &x == &one,
          &x == &(n - 1),
          &mut table_data,
          );

          // if $x \equiv -1 (\mod n)$ the input number is probably prime
          if x == n - 1 {
            return (true, table_data);
          }

          // if $x \equiv 1 (\mod n)$, then x is a factor of n
          if &x == &one {
            return (false, table_data);
          }

          k += 1;
        }

        // $a^{n-1} (\mod n) \not\equiv 1$, then by FLT, n is composite and return false.
        return (false, table_data);
      }

      pub fn test_primality_miller_rabin(n: &BigInt) -> (String, Vec<String>) {
        let mut non_witnesses: Vec<String> = Vec::new();
        let mut n_minus_one_form = String::new();
        for base in range(BigInt::from(2u64), n - 1) {
          let output = miller_rabin_test(&n, Some(&base));
          for item in output.1.iter() {
            if item.get_message().contains("Prime") {
              non_witnesses.push(base.to_string());
              if n_minus_one_form.len() == 0 {
                n_minus_one_form.push_str(&item.get_n_minus_one_form());
              }
            }
          }
        }
        (n_minus_one_form, non_witnesses)
      }
```

```
      Operations::Question3(s) => {
        let mut composites =
        list_prime_factors_in_range(&s.start, &s.end, NumCategory::Composites).1;
        // filter only odd composite numbers with only two factors
        // composites.retain(|(num, p_factors)| p_factors.len() == 2 && num % 2 != BigInt::zero());
        composites.retain(|(num, p_factors)| num % 2 != BigInt::zero());
        // take the first five elements for the test
        // let sample_data = &composites[0..5];
        println!(
        "Total Number of Odd Composites with two factors {}",
        &composites.len()
        );
        let mut json_out: BTreeMap<String, MillerRabinJson> = BTreeMap::new();
        for (num, p_factors) in composites.iter() {
          println!("Processing the number: {}", num);
          // call miller-rabin test
          let (n_minus_one_form, non_witnesses) = test_primality_miller_rabin(num);
          // Convert prime factors to String format
          let mut form = String::new();
          for (factor, exp) in p_factors {
            form.push_str(&format!("{}{} x ", factor, Superscript(exp.clone())));
          }
          let mut form = form.trim_end().to_string();
          form.pop();
          if !non_witnesses.is_empty() {
            let mr_json = MillerRabinJson::new(n_minus_one_form, form, non_witnesses);
            json_out.insert(num.to_string(), mr_json);
          }
        }

        let my_home = get_my_home()
        .unwrap()
        .unwrap()
        .to_str()
        .unwrap()
        .to_string();
        let mut output_dir = String::new();
        let mut fname = String::new();

        if cfg!(windows) {
          output_dir.push_str(&my_home);
          output_dir.push_str("\\ass1-question3");
          println!("Path = {}", &output_dir);
          fname.push_str(&output_dir);
          fname.push_str("\\");
          fname.push_str("question3.json");
        } else if cfg!(unix) {
          output_dir.push_str(&my_home);
          output_dir.push_str("/ass1-question3");
          println!("Path = {}", &output_dir);
          fname.push_str(&output_dir);
          fname.push_str("/");
          fname.push_str("question3.json");
        }
        println!("output dir: {}", &output_dir);
        if !fs::metadata(&output_dir).is_ok() {
```

```
            let _ = fs::create_dir(&output_dir);
        }
        match File::create(&fname) {
            Ok(file) => {
                println!("Output has been written to the file: {}", &fname);
                serde_json::to_writer_pretty(file, &json_out).unwrap();
            }
            Err(e) => panic!("Problem creating the file: {:?}", e),
        }
    }

```

Listing 12: Miller Rabin - Question 3

To find the numbers with Strong Liars using Miller-Rabin, execute the code using the below command:

```

    .\target\release\nt-assignments.exe question3 -s 50 -e 100

```

Listing 13: GCD Test Execution

4. (a) Choose any three elements of your set A and calculate the value of r used in the AKS primality test;

   Answer: The below code snippet calculates the r value used in AKS:

```

        ///
        /// Find smallest r such that the order of n mod r > ln(n)^2.
        ///
        pub fn findr(n: &BigInt) -> BigInt {
            let (zero, one) = (BigInt::zero(), BigInt::one());
            let mut r = BigInt::from(1u64);

            let s: f64 = abs_log(n).unwrap().pow(2);
            let s = BigInt::from(s.floor() as u64);
            let mut nex_r = true;

            while nex_r {
                r += 1;
                nex_r = false;
                let mut k = BigInt::zero();
                while &k <= &s && nex_r == false {
                    k += 1;
                    if modular_pow(n, &k, &r) == zero || modular_pow(n, &k, &
    r) == one {
                        nex_r = true;
                    }
                }
            }

            r
        }

```

Listing 14: FindR - AKS Step 2

r value calculated for selected numbers:

i. $n = 2813$
   'r' value for 2801 is = 83

ii. $n = 2837$
'r' value for 2837 is = 71

iii. $n = 2843$
'r' value for 2843 is = 101

(b) Write a single procedure that implements the AKS test using the code that we have seen; I couldn't translate the Maple code into Rust exactly as it was. I have adapted some Python code [1] I saw on the GitHub for 'r' value calculation and polynomial multiplication.

```

    ///
    /// AKS Steps
    ///
    pub enum AksSteps {
      Step1,
      Step2,
      Step3,
      Step4,
      Step5,
      Success,
    }
    ///
    /// AKS Primality test
    ///
    /// Returns boolean indicating prime or not and if the step at
 which it failed
    ///
    /// # Arguments
    /// * n: BigInt
    ///
    pub fn aks(n: &BigInt) -> (bool, AksSteps) {
      fn is_perfect_k_th_power(n: &BigInt) -> bool {
        let upper_bound = n.sqrt();
        for k in range_inclusive(BigInt::from(2u64), upper_bound) {
          let mut m = n.clone();
          let mut j = BigInt::zero();
          while &m % &k == BigInt::zero() && m > BigInt::one() {
            m /= &k;
            j += 1;
          }
          if m == BigInt::one() && j > BigInt::one() {
            return true;
          }
        }
        false
      }

      ///
      /// Find smallest r such that the order of n mod r > ln(n)^2.
      ///
      fn findr(n: &BigInt) -> BigInt {
        let (zero, one) = (BigInt::zero(), BigInt::one());
        let mut r = BigInt::from(1u64);

        let s: f64 = abs_log(n).unwrap().pow(2);
        let s = BigInt::from(s.floor() as u64);
        let mut nex_r = true;

        while nex_r {
          r += 1;
          nex_r = false;
```

```
            let mut k = BigInt::zero();
            while &k <= &s && nex_r == false {
              k += 1;
              if modular_pow(n, &k, &r) == zero || modular_pow(n, &k, &r) == one {
                nex_r = true;
              }
            }
          }

          r
        }

        // Step 1
        if is_perfect_k_th_power(n) {
          return (false, AksSteps::Step1);
        }

        let (zero, one) = (BigInt::zero(), BigInt::one());

        // Step 2
        let r = findr(n);

        // Step 3
        for a in range(BigInt::from(2u64), std::cmp::min(r.clone(), n.clone())) {
          if &a.gcd_euclid(n) > &one {
            return (false, AksSteps::Step3);
          }
        }

        // Step 4
        if !(n <= &r) {
          return (false, AksSteps::Step4);
        }

        // Step 5
        let phi_r = euler_totient_phi_counting_coprimes(&r);
        let log_r = abs_log(n).unwrap();
        let upper_bound = phi_r.sqrt() * log_r as u64;
        let mut x = Vec::<BigInt>::new();
        for a in range(BigInt::one(), upper_bound) {
          x = fastpoly(&vec![a, BigInt::one()], &n, &r);
          if x.par_iter().any(|b| b != &BigInt::zero()) {
            return (false, AksSteps::Step5);
          }
        }

        (true, AksSteps::Success)
      }
```

Listing 15: AKS Algoritm

(c) Take the elements of the set B in turn and decide how many fail the test at each of steps 1, 2, 3, 4, 5.

Answer: Below json snippet shows which numbers failed at what step:

```
{
  "Step 1": [
```

```
              2916,3025
            ],
          "Step 3": [
            2800,2802,2804,2805,2806,2807,2808,2810, 2811,
            2812,2813,2814,2815,2816,2817,2818,2820,2821,
            2822,2823,2824,2825,2826,2827,2828,2829,2830,
            2831,2832,2834,2835,2836,2838,2839,2840,2841,
            2842,2844,2845,2846,2847,2848,2849,2850,2852,
            2853,2854,2855,2856,2858,2859,2860,2862,2863,
            2864,2865,2866,2867,2868,2869,2870,2871,2872,
            2873,2874,2875,2876,2877,2878,2880,2881,2882,
            2883,2884,2885,2886,2888,2889,2890,2891,2892,
            2893,2894,2895,2896,2898,2899,2900,2901,2902,
            2904,2905,2906,2907,2908,2910,2911,2912,2913,
            2914,2915,2918,2919,2920,2921,2922,2923,2924,
            2925,2926,2928,2929,2930,2931,2932,2933,2934,
            2935,2936,2937,2938,2940,2941,2942,2943,2944,
            2945,2946,2947,2948,2949,2950,2951,2952,2954,
            2955,2956,2958,2959,2960,2961,2962,2964,2965,
            2966,2967,2968,2970,2972,2973,2974,2975,2976,
            2977,2978,2979,2980,2981,2982,2983,2984,2985,
            2986,2987,2988,2989,2990,2991,2992,2993,2994,
            2995,2996,2997,2998,3000,3002,3003,3004,3005,
            3006,3007,3008,3009,3010,3012,3013,3014,3015,
            3016,3017,3018,3020,3021,3022,3024,3026,3027,
            3028,3029,3030,3031,3032,3033,3034,3035,3036,
            3038,3039,3040,3042,3043,3044,3045,3046,3047,
            3048,3050,3051,3052,3053,3054,3055,3056,3057,
            3058,3059,3060,3062,3063,3064,3065,3066,3068,
            3069,3070,3071,3072,3073,3074,3075,3076,3077,
            3078,3080,3081,3082,3084,3085,3086,3087,3088,
            3090,3091,3092,3093,3094,3095,3096,3097,3098,
            3099,3100
            ]
        }

```

Listing 16: Miller-Rabin failues for numbers between 50 to 100

Most of the numbers failed at Step 3 while only two failed in Step 2.

5. Consider the tests we have seen so far in the module

   i. a Fermat Test calculating $a^{m-1} \pmod{m}$
   ii. a gcd test on a and m
   iii. a Miller Rabin test
   iv. Trial division/sieving methods
   v. The AKS primality test

   Thinking about factors such as:

   - the probability that the test produces a clear answer
   - the amount of work that it involves

   summarise which test you would recommend for deciding if a number is prime or not. Does the size of the target number affect your answer? Does it change for:

a. Numbers less than 10000000

b. Numbers bigger than 1000000000000

c. Numbers bigger than 10^100

Answer:

A Fermat Test calculating $a^{m-1} \pmod m$

A Fermat Primality test is a search for a Fermat Witness to certify that a given number is a composite. If it doesn't find a Fermat Witness, there is no assurance on the primality of the number. For example, Carmichael Numbers have no Fermat Witness and there are infinitely many of them. Hence if a Fermat Test outputs $a^{m-1} \equiv 1 \pmod m$ for a and m where $gcd(a, m) = 1$, there is no guarantee that the candidate number is a prime. But the probability to falsely report a number as prime is very rare if we sample enough numbers to find Fermat Witness because there are only 20138200 Carmichael numbers between 1 and 10^{21}. We can make the Fermat Test better by adding Korselt's criteria to spot Carmichael Numbers.

A gcd test on a and m

It is a probabilistic search to find a common factor between a and m. GCD test for primality is not reliable because a and m can be coprime, still composite. Euler's Totient function $\phi(n)$ returns the number of integers less than and relatively prime to n. If n large and is a composite of the form $p.q$, the $\phi(n)$ value will be almost equal to n and hence the chances that this test wrongly repost a number as prime is very high.

A Miller Rabin test

Miller-Rabin test is also a probabilistic test which looks for modular square roots other than $\pm 1 \pmod m$ to check if a number is Prime or not. It is the fastest know primality testing algorithm with high accuracy. Many cryptographic solutions use Miller-Rabin Algorithm because of its accuracy and speed. It can be used to check very large numbers. The probability of success that Miller-Rabin correctly identifying a composite number is more than 75%. It can be improved again by adding more trials to the test. Number of trials during Miller Rabin is a very important parameter as it decides the accuracy of the test. The runtime complexity of Miller-Rabin is $O(k \log^3 N)$

Trial division/sieving methods

Trial division brute-force approach checks the primality of a number by trying to find all divisors upto square root of that number. This approach is very inefficient in terms runtime. Sieving is much more efficient than trial division. Sieve of Eratostenis method first builds a list of integers and mark off even numbers, its multiples and multiples of primes in a step by step process. The time complexity of sieving is $O(n \log \log N)$ while that of trial division is $O(\sqrt{N})$

The AKS primality test

The AKS algorithm is a deterministic, polynomial time primality checking method. It's time complexity[3] is $O(r^{3/2}(\log n)^3)$

For the second part of the problem, I suggest the Miller Rabin Algorithm for any input size. It is faster because of it probabilistic in nature. By adding enough trials to primality checking, we can make sure that the probability of it not correctly identifying a composite number is reduced. The implementation is also easy. Almost all cryptographic libraries such as OpenSSL, GMP, GnuPG, etc use Miller Rabin Primality tests. Deterministic primality tests are far too slow for very large numbers.

References

[1] *https://github.com/Ssophoclis/AKS-algorithm/blob/master/AKS.py*

[2] *https://exploringnumbertheory.wordpress.com/tag/fermat-primality-test/*

[3] *https://en.wikipedia.org/wiki/AKS_primality_test*

[4] Andrew Granville *It Is Easy To Determine Whether A Given Integer Is Prime*, Bulletin (New Series) Of The American Mathematical Society, 2004

[5] Jake Massimo *An Analysis of Primality Testing and Its Use in Cryptographic Applications*, PhD Thesis, Royal Holloway,University of London, 2020