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:

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

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

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

```rust

    /// 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;
    /// 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 $n = p.q$

```
    ///
    /// 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
    /// * `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 $n = 2881$

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

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;
```

```
58            while &r > &zero {
59              // let q = &a / &b;
60              r = &a % &b;
61
62              if &r != &zero {
63                a = b;
64                b = r.clone();
65              }
66            }
67
68            gcd = b;
69          }
70
71          gcd
72        }
73      }
74
```

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:

```
1
2       /// Returns a list of Carmichael Numbers (Absolute Pseudoprimes) in a range using FLT or
      Korselt's criterion
3       ///
4       /// # Arguments
5       /// * start: BigInt
6       /// * end: BigInt
7       /// * f: a function pointer to either primality::carmichael_nums_korselt or primality::
      carmichael_nums_flt
8       /// # Examples
9       /// ```
10      /// use crate::presets::list_carmichael_nums;
11      /// let carmichael_nums = list_carmichael_nums(&start, &end, carmichael_nums_flt);
12      /// ```
13      ///
14      pub fn list_carmichael_nums(start: &BigInt, end: &BigInt, f: fn(&BigInt) -> bool) -> (String,
       Vec<(BigInt, Vec<(BigInt, usize)>)>) {
15        // Get all the composite numbers in the range
16        let composites = list_prime_factors_in_range(start, end, NumCategory::Composites).1;
17
18        // Searching for Carmichael numbers in parallel
19        let carmichael_nums = composites
20        .par_iter()
21        .filter(|x| f(&x.0) == true)
22        .map(|x| x.clone())
23        .collect::<Vec<(BigInt, Vec<(BigInt, usize)>)>>();
24
25        // Format the data for printing
26        let mut table_data: Vec<NumFactorTable> = Vec::new();
27        for item in carmichael_nums.iter() {
28          let mut form: String = String::new();
29          format_prime_factors_print(&item.0, &item.1, &mut form, &mut table_data);
30        }
31
32        let mut table1 = Table::new(table_data);
33        table1.with(STYLE_2);
34
35        let output1 = table1.to_string();
36        (output1, carmichael_nums)
37      }
38
39      ///
40      /// Carmichael Numbers using FLT
41      /// n: a composite number
42      ///
43      pub fn carmichael_nums_flt(n: &BigInt) -> bool {
44        let n_minus_one = n - 1;
45        // Get all the coprime numbers less than `n`
46        let coprimes_n = coprime_nums_less_than_n(n);
47
48        // Search for Fermat Witnesses. A Fermat Witness will yeild a^{n-1} ≢ 1(mod n)
49        let fermat_witnesses = coprimes_n
50        .par_iter()
51        .filter(|x| modular_pow(&x, &n_minus_one, n) != BigInt::one())
52        .map(|x| x.clone())
53        .collect::<Vec<BigInt>>();
54
55        // No Fermat Witness means n is a Carmichael Number
56        fermat_witnesses.len() == 0
57      }
58
```

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

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:

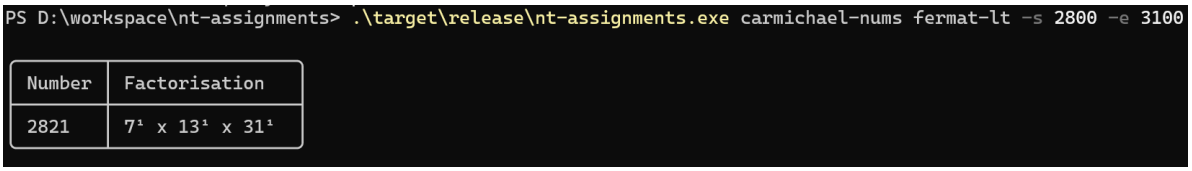

Figure 5: Carmichael Number using FLT- Example result

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. $p|n \implies (p-1)|(n-1)$.

(c) $|\mathbf{G}| = \mathbf{128}$.
The prime factorization is given by $128 = 2^7$. It's a p-group and hence the center of the group Z(G) is not trivial. Also Z(G) is a proper normal subgroup of G. By Cauchy's theorem, if $|G| = n$ and if $n = pm$ where p is a prime and $p|m$, then $\exists g \in G$ such that $g^p = e \implies |g| = p$

In our case, we can write $|G|$ as $128 = 2 \times 2^6$. So $p = 2$ and $m = 2^6$. This implies that we can generate the entire group using the single element 'g', i.e,
$G = \langle g | g^{128} = e \rangle$ and this is cyclic. Every cyclic group is abelian and hence Z(G) = G, but the order is not prime and hence **not simple**

(d) $|\mathbf{G}| = \mathbf{129}$. Upon prime factorization, $129 = 3 \times 43$
.
By Lagrange's theorem, the order of a subgroup must divide the order of the group. The factors of 129 are $\{1, 3, 43, 129\}$. For a simple group, the only normal subgroups are the trivial group $\{e\}$ and the improper subgroup G itself. So subgroups of order 1 and 129 are out of question here. We need to verify if subgroups of order 3 and 43 are normal.

Let's verify if we have a Sylow 43-subgroup in G. By Sylow's third theorem,
$n_{43} \equiv 1 (\text{mod } 43)$ and $n_{43}|3$.
We have $n_{43} = 1$ satisfies the condition. That means there exists a unique proper normal Sylow 43-subgroup for a group of order 129. So the group is **not simple.**

(e) $|\mathbf{G}| = \mathbf{130}$.

$|G| = 2 \times 5 \times 13$. If we consider Sylow 13-subgroup in G, by Sylow's third theorem,
$n_{13} \equiv 1 (\text{mod } 13)$ and $n_{13}|10$.
Only $n_{13} = 1$ can satisfies these two conditions. That means there exists a unique proper normal Sylow 13-subgroup for a group of order 130. So the group is **not simple.**

(f) $|\mathbf{G}| = \mathbf{131}$.
131 is a prime number. By Lagrange's theorem, the order of a subgroup must divide the order of the group. The only factors of 131 are 1 and 131 itself and hence there are no proper normal subgroups for a group or order 131. So the group is **Simple.**

(g) $|\mathbf{G}| = \mathbf{132}$.
The prime factorization of $|G|$ is given by $|G| = 2^2 \times 3 \times 11$. We will consider Sylow 11-subgroups first.
The constraints are $n_{11} \equiv 1 (\text{mod } 11)$ and $n_{11}|12$ and $n_{11} = \{1, 12\}$ satisfies these constraints. If $n_{11} = 1$, then the group has a proper normal Sylow 11-subgroup and hence G is not simple. If $n_{11} = 12$, there are 12 subgroups with 10 elements of order 11 in each (the identity element is shared). So The total number of elements of order 11 in G is 120.

Now if we consider Sylow 3-subgroups, the constraints are $n_3 \equiv 1(\text{mod } 3)$ and $n_3|44$. $n_3 = \{1, 4, 22\}$ satisfies these constraints. Now if we consider $n_3 = 1$, then there exists a proper normal Sylow 3-subgroup and hence G is not simple. If we consider $n_3 = 4$, there exist 4 Sylow 3-subgroups. Hence the total number of elements in the group now is $120 + 4 \times 2 = 128$. Only 4 elements remaining and a Sylow 2-subgroup of order 4 will fill that. Then the Sylow 2-subgroup is a unique proper normal subgroup hence G is not simple. If we consider $n_3 = 22$, then the total number of elements becomes $120 + 22 \times 2 = 164$ which is greater than 132 and hence $n_3 = 22$ is not possible.
We will now consider the Sylow 2-Subgroups. The constraints are $n_2 \equiv 1(\text{mod } 2)$ and $n_2|33$. $n_2 = \{1, 3, 11, 33\}$ satisfies these constraints. If $n_2 = 1$, then there exists a proper normal Sylow 2-subgroup of order 4 and hence the group G is not simple. $n_2 = \{3, 11, 33\}$ will not tally to 132 and hence those values are not possible. So a group of order $|G| = 132$ is not a simple group

4. Let G be a group and suppose H_1 and H_2 are subgroups of G such that there exists $g \in G$ such that $H_1 = H_2 = \{h^g : h \in H\}$

   (a) Show that $H_1 \cong H_2$.

   Answer: Given $H_1, H_2 \leq G$
   Let σ be the isomorphic mapping from H_1 to H_2. Then σ is given by: $\sigma_g : H_1 \to H_2$ and is defined as $\sigma_g : h \mapsto ghg^{-1}$, $\forall h \in H_1$. Let $h_1, h_2 \in H_1$. Then for an isomorphism, the following constraint must be satisfied along with the mapping σ being bijective.
   $\sigma(h_1h_2) = \sigma(h_1)\sigma(h_2)$
   LHS: $\sigma(h_1h_2) = g(h_1h_2)g^{-1}$ RHS: $\sigma(h_1)\sigma(h_2) = gh_1g^{-1}gh_2g^{-1} = gh_1eh_2g^{-1} = gh_1h_2g^{-1}$
   Hence $LHS = RHS$ and σ is a homomorphism.
   Proof for Bijection: We can prove it by showing σ_g has two sided inverse, that's if $\sigma_g^{-1} = h :\mapsto g^{-1}hg$ then we need to prove $\sigma_g^{-1}(\sigma_g(h)) = h$, $\forall h \in H1$ and $\sigma_g(\sigma_g^{-1}(h)) = h$, $\forall h \in H1$
   1. $\sigma_g^{-1}(\sigma_g(h)) = \sigma_g^{-1}(ghg^{-1}) = g^{-1}(ghg^{-1})g = h$
   2. $\sigma_g(\sigma_g^{-1}(h)) = \sigma_g(g^{-1}hg) = g(g^{-1}hg)g^{-1} = h$
   Hence the mapping is bijective. $\therefore H_1 \cong H_2$

   (b) Now suppose that G is finite and that $P, P' \in Syl_p(G)$. Explain why P and P' are isomorphic.

   Answer: If we have $P, P' \in Syl_p(G)$ then they are conjugates in G by Sylow's second theorem. i.e, $\exists g \in G$ such that $P' = gPg^{-1}$. Conjugate groups are always isomorphic.

References

[1] C R Jordan & D A Jordan *MODULAR MATHEMATICS Groups* .

[2] Dr. Ben Fairbairn *GROUP THEORY Solutions to Exercises.*

[3] *https://yutsumura.com/sylows-theorem-summary/*