Assignment

Problems identified by x.y(z) denote the problem “y”, in chapter “x” of the textbook, with part “z”. If “z” is not noted, then the entire problem is required.

Due Jan 19

1.2 Big-Oh comparisons
- Write a function to factor integers into their prime factors. Use the brute force algorithm of dividing by all integers from 2 to ceil(sqrt(N)). Use `long long int` type for your integers. Remember that some factors might be repeated. For example, 4 = 2 * 2. Test for correctness for numbers 2 - 1000. Consider the `ldiv()` function from stdlib. If the number is still larger than 1 when ceil(sqrt(N)) is reached, then the number is a prime factor. There can’t be more than 1 factor larger than sqrt(N). Do you see why?

Due Jan 22

1.7 Big-Oh analysis of an algorithm
1.8 The correctness is not a proof but a justification. Look at the way multiplication is justified in the text, and follow a similar approach.
- Use your factorization function to plot a runtime complexity chart for the factorization. Use the number of significant bits for the X-axis and cputime for the Y-axis. Add normalized theoretical curves for log(N), N, N*log(N), N^2, N^3 and 2^N. Factor the largest 1024 numbers in each bit size, unless there are fewer than 1024, then factor them all. Be sure that you don’t time the checking, only the factoring. Get at least to 50 bit numbers. Try to get all the way to 63 bit numbers.

Due Jan 24

1.11 Apply modular exponentiation algorithm
1.12 Apply modular exponentiation algorithm. That is 2^(2^2006).
1.17 It may help you to write out the two algorithms
- Write a new function that receives a `long long int` and returns `true` if the number is prime, and `false` if it is not. Use the factoring code as a basis for your algorithm. If the number only has 1 prime factor (itself), then it is prime. Use this function to find the 1024 largest primes in any bit size of numbers. Verify that your code works for up to at least 50 bit numbers. Add normalized theoretical curves for log(N), N, N*log(N), N^2, N^3 and 2^N.

Due Jan 26

1.20 Apply extended-Euclid algorithm. Report the values of x and y for each problem.
1.22 This is a short proof
1.39 That is a^(b^c)
- (NOT SPRING 2019) Use your prime number testing code to create a plot of the runtime complexity for prime number verification. On the X-axis use the number of significant bits. On the Y-axis, use the cputime per number checked. That’s all numbers checked, not just the prime numbers. Find the primes in at least the 1024 largest numbers in each bit group to gather your statistics. Add the usual collection of normalized theoretical curves to your plot.

Due Jan 29

1.27 Apply RSA algorithm. Only find d.
1.28 Apply RSA algorithm
- Implement the ` primality2 ` function from the text book, for `long long int`s. Use this function to find the 1024 largest primes in any bit size of numbers. Verify that your code works for up to at least 31 bit numbers. You can compare the results of your first primality test to the results of this test. (Why only 31 bit numbers?) This work requires that you verify your function is correct. It does not require the timing and charting of complexity. Note 1 This will require implementing modular exponentiation. Note 2 The values chosen for [a] in the test must be relatively prime to the potential prime number being tested. Note 3 Be sure to test your `gcd` and `modexp` functions for correctness, in the case of small and large input values.

Due Jan 31
1.27 Apply RSA algorithm. Find the encryption of M. Use the value of d found previously.

1.42 Apply theory of RSA algorithm

Use your primality2 prime number testing code to create a plot of the runtime complexity for this version of prime number verification. On the X-axis use the number of significant bits. On the Y-axis, use the cputime per number checked. That’s all numbers checked, not just the prime numbers. Find at least the 1024 largest primes in any bit size of numbers to gather your statistics. Add the usual collection of normalized theoretical curves to your plot. Also include the runtimes for your original brute-force prime number checker. Is the result what you expected? Explain it.

(Not required, only for fun): Challenge: Find as many 31-bit Carmichael numbers as you can. That is, find Carmichael numbers in the range from \((2^{31} - 1)\) down to \((2^{30})\), inclusive. Submit a text file with the numbers 1 per line from largest to smallest. The student with the most Carmichael numbers wins the respect of every one in the class. False positives will disqualify the submission. False negatives will not disqualify the submission.

**Submission**

- At the beginning of class on the due dates, submit paper copies of your solutions.