**Structures and Interpretation of Computer Program**

**Exercise Chapter 1.2 Name:** Wan Huzaifah bin Wan Azhar

**Exercise 1.2.5 Testing for Primality**

1. Find smallest divisor for 199, 1999, 19999

(small divisor 199)  
(find-divisor 199 2)   
(find-divisor 199 3)   
(find-divisor 199 4)   
(find-divisor 199 5)   
(find-divisor 199 6)   
(find-divisor 199 7)   
(find-divisor 199 8)   
(find-divisor 199 9)   
(find-divisor 199 10)   
(find-divisor 199 11)   
(find-divisor 199 12)   
(find-divisor 199 13)   
(find-divisor 199 14)   
(find-divisor 199 15)  
  
# ( > (square 15 ) 199) 199  
  
199 – no divisor

--  
(small-divisor 1999)  
(find-divisor 1999 2)   
(find-divisor 1999 3)   
(find-divisor 1999 4)   
(find-divisor 1999 5)   
(find-divisor 1999 6)   
(find-divisor 1999 7)   
(find-divisor 1999 8)   
(find-divisor 1999 9)   
(find-divisor 1999 10)   
(find-divisor 1999 11)   
(find-divisor 1999 12)   
(find-divisor 1999 13)   
(find-divisor 1999 14)  
…  
(find-divisor 1999 45)  
# ( > (square 45 ) 1999)   
1999 – no divisor

--

(small-divisor 19999)  
(find-divisor 19999 2)   
(find-divisor 19999 3)   
(find-divisor 19999 4)   
(find-divisor 19999 5)   
(find-divisor 19999 6)   
(find-divisor 19999 7)

#(divides? 19999 7)

7 – divides 19999

1. Create prime-iteration and compare runtime

(define (even x)

(= (/ x 2) 0))

(define (smallest-divisor n)

(find-divisor n 2))

(define (find-divisor n test-divisor)

(cond ((> (square test-divisor) n) n)

((divides? test-divisor n) test-divisor)

(else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b)

(= (remainder b a) 0))

(define (prime? n)

(= n (smallest-divisor n)))

(define (timed-prime-test n)

(newline)

(display n)

(start-prime-test n (runtime)))

(define (start-prime-test n start-time)

(if (prime? n)

(report-prime (- (runtime) start-time))))

(define (report-prime elapsed-time)

(display " \*\*\* ")

(display elapsed-time))

(define (search-for-prime-iter b n bd)

(timed-prime-test b)

(cond ( (or (> n 2) (> b bd)) (display " - Finished"))

((prime? b) (search-for-prime-iter (+ b 1) (+ n 1) bd))

(else (search-for-prime-iter (+ b 1) n bd))))

(define (search-for-prime b bd)

(search-for-prime-iter b 0 bd))

(search-for-prime 1000000000 10000000000)

(search-for-prime 10000000000 100000000000)

(search-for-prime 100000000000 1000000000000)

Output: <https://pastebin.com/isYb9B0v>

From the output, un searching for prime of 10^9, we can see that it took around 0.03 of runtime and searching for prime of 10^10 yields 0.07 of runtime which is roughly factor of 3 (sqrt(10) ~ 3). Runtime of searching for prime of 10^11 resulted in around 0.24 which is also the factor of runtime of 10^10. All in all, the result shows that the order of growth is sqrt(n).

1. Simplify smallest-divisor with next function

(define (next x)

(cond ((= x 2) 3)

(else (+ x 2))))

(define (even x)

(= (/ x 2) 0))

(define (smallest-divisor n)

(find-divisor n 2))

(define (find-divisor n test-divisor)

(cond ((> (square test-divisor) n) n)

((divides? test-divisor n) test-divisor)

(else (find-divisor n (next test-divisor)))))

|  |  |  |
| --- | --- | --- |
| N | Run-time | Run-time with next |
| 10^9 | 0.03 | 0.01 |
| 10^10 | 0.07 | 0.049 |
| 10^11 | 0.24 | 0.14 |

The table shows that when the algorithm is replaced with next function, the run-time decrease by around half (not really half, but around 1.5), which shows that if you skip or reduce the step, the run-time also decrease.

1. Replace prime? with fast-prime? And analyze the result

(define (expmod base exp m)

(cond ((= exp 0) 1)

((even? exp)

(remainder (square (expmod base (/ exp 2) m))

m))

(else

(remainder (\* base (expmod base (- exp 1) m))

m))))

(define (fermat-test n)

(define (try-it a)

(= (expmod a n n) a))

(try-it (+ 1 (random (- n 1)))))

(define (fast-prime? n times)

(cond ((= times 0) true)

((fermat-test n) (fast-prime? n (- times 1)))

(else false)))

(define (next x)

(cond ((= x 2) 3)

(else (+ x 2))))

(define (even x)

(= (/ x 2) 0))

(define (smallest-divisor n)

(find-divisor n 2))

(define (find-divisor n test-divisor)

(cond ((> (square test-divisor) n) n)

((divides? test-divisor n) test-divisor)

(else (find-divisor n (next test-divisor)))))

(define (divides? a b)

(= (remainder b a) 0))

(define (prime? n)

(= n (smallest-divisor n)))

(define (timed-prime-test n)

(newline)

(display n)

(start-prime-test n (runtime)))

(define (start-prime-test n start-time)

(if (fast-prime? n 100)

(report-prime (- (runtime) start-time))))

(define (report-prime elapsed-time)

(display " \*\*\* ")

(display elapsed-time))

(define (search-for-prime-iter b n bd)

(timed-prime-test b)

(cond ( (or (> n 2) (> b bd)) (display " - Finished"))

((prime? b) (search-for-prime-iter (+ b 1) (+ n 1) bd))

(else (search-for-prime-iter (+ b 1) n bd))))

(define (search-for-prime b bd)

(search-for-prime-iter b 0 bd))

(search-for-prime 1000000000 10000000000)

(search-for-prime 10000000000 100000000000)

(search-for-prime 100000000000 1000000000000)

(search-for-prime 1000000000000 100000000000000)

|  |  |  |
| --- | --- | --- |
| N | Run-time | Run-time with fast-prime |
| 10^9 | 0.03 | 0.009 |
| 10^10 | 0.07 | 0.01 |
| 10^11 | 0.24 | 0.009 |

Replacing prime? with fast-prime increase the speed by almost factor of 3. This is because by replacing prime function with fast-prime, the primality of the number is tested using O(log n) of fermat-test by 100 times. Overall, the run-time of all the N are close because the algorithm test it by fixed amount of 100 times by growth of log n.

1. By replacing expmod with fast-expt, the algorithm calculates the exponential of n first before calculating the remainder. This approach can work if the number is small, e,g, 5. But, if the number is big like 10^10, calculating exponential of it still takes time as it successively square each number one after another. Expmod avoid this problem by squaring and giving back the remainder of the number so that the calculating is easy as the number is small.
2. By using primitive multiplication, the algorithm will call expmod two times instead of calculating the value of expmod and squaring it. This may seem small but when calling expmod two times, it will fill up the stack two times until it reaches a stopping point of exp = 0, thus generating tree like algorithm rather than linear algorithm. As such, like the exercise said, it transforms log n process into n process.
3. Testing Carmichael number for primality

(define (expmod base exp m)

(cond ((= exp 0) 1)

((even? exp)

(remainder (square (expmod base (/ exp 2) m))

m))

(else

(remainder (\* base (expmod base (- exp 1) m))

m))))

(define (fermat-test n m)

(define (try-it a)

(= (expmod a n n) a))

(try-it m))

(define (fast-prime? n times)

(cond ((= times 0) true)

((fermat-test n times) (fast-prime? n (- times 1)))

(else false)))

;Do a fermat-test of n value by n-1 times

(define (fermat-test-mic n)

(fast-prime? n (- n 1)))

(if (fermat-test-mic 560) (display "560 is prime"))

(newline)

(if (fermat-test-mic 677) (display "677 is prime"))

(newline)

(if (fermat-test-mic 561) (display "561 is prime but it is not"))

(newline)

(if (fermat-test-mic 1105) (display "1105 is prime but it is not"))

(newline)

(if (fermat-test-mic 1729) (display "1729 is prime but it is not"))

(newline)

(if (fermat-test-mic 2465) (display "2465 is prime but it is not"))

(newline)

(if (fermat-test-mic 2821) (display "2821 is prime but it is not"))

(newline)

(if (fermat-test-mic 6601) (display "6601 is prime but it is not"))

Output:

