1. Problema: Scrieți un cod în Racket pentru a găsi toate numerele prime dintr-o listă de numere.

```
(define (prime-numbers lst)
  (filter prime? lst))
(prime-numbers '(2 3 4 5 6 7 8 9 10))
```

2. Problema: Scrieți un cod în Racket pentru a găsi suma tuturor numerelor pare dintr-o listă de numere.

```
(define (sum-even-numbers lst)
  (foldl (lambda (x acc) (if (even? x) (+ x acc) acc)) 0 lst))
(sum-even-numbers '(1 2 3 4 5 6 7 8 9 10))
```

3. Problema: Scrieți un cod în Racket pentru a verifica dacă o listă de numere este palindrom.

```
(define (palindrome? lst)
  (equal? lst (reverse lst)))
(palindrome? '(1 2 3 2 1))
```

4. Problema: Scrieți un cod în Racket pentru a găsi cel mai mare număr dintr-o listă de numere.

```
(define (max-number lst)
  (apply max lst))
(max-number '(5 2 9 1 7 3))
```

5. Problema: Scrieți un cod în Racket pentru a calcula suma cifrelor unui număr dat.

```
(define (sum-of-digits n)
  (apply + (map (lambda (x) (- (char->integer x) (char->integer #\0))) (string->list (number->string n)))))
(sum-of-digits 12345)
```

6. Problema: Scrieți un cod în Racket pentru a verifica dacă un cuvânt este un palindrom.

```
(define (palindrome-word? word)
  (equal? word (reverse word)))
(palindrome-word? "radar")
```

7. Problema: Scrieți un cod în Racket pentru a verifica dacă toate cuvintele dintr-o listă au aceeași lungime.

```
(define (same-length? lst)
```

```
  (apply = (map string-length lst)))
(same-length? '("abc" "def" "ghi"))
```

8. Problema: Scrieți un cod în Racket pentru a găsi media aritmetică a unei liste de numere.

```
(define (average lst)
  (/ (apply + lst) (length lst)))
(average '(1 2 3 4 5))
```

9. Problema: Scrieți un cod în Racket pentru a găsi produsul tuturor numerelor impare dintr-o listă de numere.

```
(define (product-of-odd-numbers lst)
  (foldl (lambda (x acc) (if (odd? x) (* x acc) acc)) 1 lst))
(product-of-odd-numbers '(1 2 3 4 5))
```

10. Problema: Scrieți un cod în Racket pentru a găsi toate cuvintele dintr-un text dat care încep cu litera "a".

```
(define (words-starting-with-a text)
  (filter (lambda (word) (char=? (string-ref word 0) #\a)) (string-split text)))
(words-starting-with-a "ana are mere si pere")
```

11. Problema: Scrieți un cod în Racket pentru a găsi cel mai mic număr par dintr-o listă de numere.

```
(define (min-even-number lst)
  (apply min (filter even? lst)))
(min-even-number '(3 1 4 1 5 9 2 6 5))
```

12. Problema: Scrieți un cod în Racket pentru a verifica dacă toate cuvintele dintr-o listă sunt scrise cu litere mici.

```
(define (all-lowercase? lst)
  (every (lambda (word) (string=? word (string-downcase word))) lst))
(all-lowercase? '("apple" "banana" "pear"))
```

13. Problema: Scrieți un cod în Racket pentru a găsi numerele prime mai mici decât un număr dat.

```
(define (primes-less-than n)
  (filter prime? (range 2 n)))
```

(primes-less-than 20)

14. Problema: Scrieți un cod în Racket pentru a verifica dacă o listă de numere este strict crescătoare.

15.Problema: Scrieți un cod în Racket pentru a găsi diferența maximă dintre două numere consecutive dintr-o listă de numere.

```
(define (max-consecutive-difference lst)
  (apply max (map (lambda (x y) (abs (- x y))) lst (cdr lst))))
(max-consecutive-difference '(1 5 2 9 3))
```

Problema: Scrieți un cod în Racket pentru a găsi suma pătratelor numerelor pozitive dintr-o listă de numere.

```
(define (sum-of-positive-squares lst)
  (foldl (lambda (x acc) (if (> x 0) (+ (* x x) acc) acc)) 0 lst))
(sum-of-positive-squares '(-2 3 -1 4 -5))
```

Problema: Scrieți un cod în Racket pentru a găsi toate permutările unei liste de numere.

```
(require math.combinatorics)
(define (permutations lst)
  (list->vector (permutations-of lst)))
(permutations '(1 2 3))
```

Problema: Scrieți un cod în Racket pentru a verifica dacă o listă de numere conține duplicate.

```
(define (has-duplicates? lst)
  (not (equal? (length lst) (length (remove-duplicates lst)))))
(has-duplicates? '(1 2 3 4 5 2))
```

Problema: Scrieți un cod în Racket pentru a verifica dacă toate cuvintele dintr-un text dat sunt unice.

```
(define (all-unique-words? text)
  (let ((words (string-split text)))
    (equal? (length words) (length (remove-duplicates words)))))
(all-unique-words? "apple banana apple pear")
```

Problema: Scrieți un cod în Racket pentru a verifica dacă toate numerele dintr-o listă sunt impare.

```
(define (all-odd-numbers? lst)
  (every odd? lst))
(all-odd-numbers? '(1 3 5 7 9))
```

Problema: Scrieți un cod în Racket pentru a găsi numărul maxim de caractere dintr-o listă de cuvinte.

```
(define (max-characters lst)
  (apply max (map string-length lst)))
(max-characters '("apple" "banana" "pear"))
```

Problema: Scrieți un cod în Racket pentru a găsi suma tuturor numerelor pozitive dintr-o listă de numere.

```
(define (sum-positive-numbers lst)
  (foldl (lambda (x acc) (if (> x 0) (+ x acc) acc)) 0 lst))
(sum-positive-numbers '(-2 3 -1 4 -5))
```

Problema: Scrieți un cod în Racket pentru a găsi toate numerele dintr-o listă care sunt patrate perfecte.

```
(define (perfect-squares lst)
  (filter (lambda (x) (= (sqrt x) (round (sqrt x)))) lst))
(perfect-squares '(1 2 3 4 5 6 7 8 9 10))
```

Problema: Scrieți un cod în Racket pentru a găsi diferența minimă dintre două numere consecutive dintr-o listă de numere.

```
(define (min-consecutive-difference lst)
  (apply min (map (lambda (x y) (abs (- x y))) lst (cdr lst))))
(min-consecutive-difference '(1 5 2 9 3))
```

Problema: Scrieți un cod în Racket pentru a verifica dacă toate cuvintele dintr-o listă sunt anagrame ale unui cuvânt dat.

```
(define (anagrams? word lst)
  (every (lambda (w) (anagram? word w)) lst))
(anagrams? "listen" '("silent" "enlist" "tinsel"))
```