Discrete Functions Essentials

An essential guide to discrete functions for students of computer science, mathematics, and engineering. This cheat sheet covers core definitions, types of functions, sequences, special functions, and asymptotic growth, providing clear examples and key insights.

Foundational Concepts

FUNCTION BASICS

Definition: A discrete function f: A \to B maps each element from a discrete domain A to exactly one element in a discrete codomain B.

Notation:

- A: Domain (set of inputs)
- B: Codomain (set of possible outputs)
- f(a): Image of a
- Range: {f(a) \mid a \in A} (subset of codomain)

One-to-One (Injective): Every distinct element in the domain maps to a distinct element in the codomain.

- No two domain elements map to the same codomain element.
- \forall a_1, a_2 \in A, f(a_1) = f(a_2) \implies a_1 = a_2

Example: f: \mathbb{Z} \to \mathbb{Z}, f(x) = x+1.

- If x_1+1 = x_2+1, then x_1 = x_2. This is one-to-one.

Non-example: g: \mathbb{Z} \to \mathbb{Z}, g(x) = x^2.

- g(2)=4, g(-2)=4. Not one-to-one.

Onto (Surjective): Every element in the codomain is the image of at least one element in the domain.

- The range of the function is equal to its codomain.
- \forall b \in B, \exists a \in A \text{ such that } f(a) = b

Example: f: \mathbb{Z} \to \mathbb{Z}, f(x) = x-3.

- For any y \in \mathbb{Z}, we can find x = y+3 \in \mathbb{Z} such that f(x)=y. This is onto.

Non-example: g: \mathbb{Z} \to \mathbb{N}_0, g(x) = |x|.

- Codomain is non-negative integers. Only non-negative integers are in the range. This is onto.

Bijective (One-to-One Correspondence): A function that is both one-to-one and onto.

- Each element in the domain maps to exactly one unique element in the codomain, and every element in the codomain has exactly one pre-image in the domain.

Example: f: \mathbb{Z} \to \mathbb{Z}, f(x) = x+5.

- This function is both one-to-one (as shown with x+1) and onto (as shown with x-3). Thus, it's bijective.

Importance: Bijective functions have inverse functions.

Function Composition: (g \circ f)(x) = g(f(x)). Applies f first, then g.

- Domain of g must contain the range of f.

Example: f(x) = x+1, g(x) = x^2.

- (g \circ f)(x) = (x+1)^2
- (f \circ g)(x) = x^2+1
- Note: (g \circ f)(x) \ne (f \circ g)(x) usually.

Key Insight: Understanding if a function is one-to-one, onto, or bijective is crucial for inverse functions, counting arguments (cardinality), and cryptographic applications.

SEQUENCES & RECURSIVE FUNCTIONS

Sequence Definition: An ordered list of elements. Can be finite or infinite.

- Often defined by a function a: \mathbb{N} \to S, where a_n is the n-th term.

Example (Arithmetic): Sequence: 2, 5, 8, 11, \dots

- a_1 = 2, Common difference d = 3.
- a_n = 2 + (n-1)3

Example (Geometric): Sequence: 3, 6, 12, 24, \dots

- a_1 = 3, Common ratio r = 2.
- a_n = 3 \cdot 2^{n-1}

Fibonacci Sequence: A classic example of a recurrence relation.

- F_n = F_{n-1} + F_{n-2} for n \ge 2
- Initial conditions: F_0 = 0, F_1 = 1
- Sequence: 0, 1, 1, 2, 3, 5, 8, 13, \dots

Iteration vs. Recursion (Programming Context):

- **Iteration:** Uses loops (for, while) to repeat a process.

```
def factorial_iter(n):
    res = 1
    for i in range(1, n + 1):
        res *= i
    return res
```

Arithmetic Sequence: Each term is found by adding a constant (common difference d) to the previous term.

- Formula: a_n = a_1 + (n-1)d

Geometric Sequence: Each term is found by multiplying the previous term by a constant (common ratio r).

- Formula: a_n = a_1 \cdot r^{n-1}

Recurrence Relation: Defines a term in a sequence based on one or more preceding terms.

- Requires initial conditions (base cases).

Solving Recurrence Relations (Brief):

1. **Iteration:** Compute terms until a pattern emerges.
2. **Characteristic Equation:** For linear homogeneous relations.
3. **Generating Functions:** A more general approach.

- **Recursion:** A function calls itself to solve smaller subproblems.

```
def factorial_rec(n):
    if n == 0:
        return 1
    else:
        return n * factorial_rec(n - 1)
```

Tail Recursion: A special form where the recursive call is the last operation. Can often be optimized by compilers into iteration.

Common Pitfall: Forgetting base cases in recurrence relations or recursive functions leads to infinite loops or incorrect results.

Special Functions & Asymptotic Analysis

PIECEWISE & STEP FUNCTIONS

Piecewise Function: Defined by multiple sub-functions, each applied to a different interval of the domain.

- Syntax: Defined with curly braces and conditions.

Example:
f(x) = \begin{cases} x^2 & \text{if } x < 0 \\ x & \text{if } x \ge 0 \end{cases}

- f(-3) = (-3)^2 = 9
- f(5) = 5

Floor Function (Greatest Integer Function): \lfloor x \rfloor

- Returns the largest integer less than or equal to x.
- Rounds down to the nearest integer.

Examples:

- \lfloor 3.14 \rfloor = 3
- \lfloor 7 \rfloor = 7
- \lfloor -2.5 \rfloor = -3
- \lfloor 0.99 \rfloor = 0

Ceiling Function (Least Integer Function): \lceil x \rceil

- Returns the smallest integer greater than or equal to x.
- Rounds up to the nearest integer.

Examples:

- \lceil 3.14 \rceil = 4
- \lceil 7 \rceil = 7
- \lceil -2.5 \rceil = -2
- \lceil 0.01 \rceil = 1

Properties of Floor/Ceiling:

- x-1 < \lfloor x \rfloor \le x
- x \le \lceil x \rceil < x+1
- \lfloor x+n \rfloor = \lfloor x \rfloor + n for integer n
- \lceil x+n \rceil = \lceil x \rceil + n for integer n

Applications: Used in computer science for array indexing, memory allocation, and time complexity analysis.

Key Insight: Floor and ceiling functions are crucial for discretizing continuous values, which is fundamental in many computational algorithms where only integer quantities are meaningful (e.g., number of blocks, number of iterations).

MODULAR ARITHMETIC FUNCTIONS

Modulo Operator (mod): a \pmod n

- Returns the remainder when integer a is divided by integer n (where n > 0).
- The result is always in the range [0, n-1].
- a = qn + r, where r = a \pmod n

Function: f(x) = x \pmod n

- Domain: Integers \mathbb{Z}
- Codomain: {0, 1, \dots, n-1}
- This function maps any integer to one of the n possible remainders.

Examples:

- 17 \pmod 5 = 2 (since 17 = 3 \cdot 5 + 2)
- 25 \pmod 5 = 0 (since 25 = 5 \cdot 5 + 0)
- -8 \pmod 5 = 2 (since -8 = -2 \cdot 5 + 2)

Note on Negative Numbers: Different programming languages might handle negative numbers differently for the `%` operator. In mathematics, the result of a \pmod n is always non-negative.

Congruence Relation: a \equiv b \pmod n

- Means a and b have the same remainder when divided by n.
- Equivalent to saying n divides (a-b), i.e., a-b = kn for some integer k.

Example:

- 17 \equiv 2 \pmod 5
- -8 \equiv 2 \pmod 5
- 10 \equiv 0 \pmod 5

Applications in Computer Science:

- **Hashing:** Maps data to fixed-size array indices (hash table buckets). `index = hash(key) % array_size`
- **Cryptography:** RSA, Diffie-Hellman rely heavily on modular exponentiation and modular inverse.
- **Cyclic Data Structures:** Ring buffers, circular arrays.
- **Time and Date Calculations:** E.g., finding the day of the week.

Example (Hashing): If a hash table has 10 buckets, and `hash(key)` returns 12345, then `12345 % 10 = 5`. The item goes into bucket 5.

Common Pitfall: Forgetting that modulo results are always non-negative in pure mathematics, unlike some programming language implementations where `(-a) % n` might be negative.

GROWTH OF FUNCTIONS

Asymptotic Analysis: Studies the behavior of functions as their input (usually n) approaches infinity.

- Crucial for analyzing algorithm efficiency (time and space complexity).

Big-O Notation (O(g(n))): Upper Bound

- f(n) \in O(g(n)) if there exist positive constants c and n_0 such that 0 \le f(n) \le c \cdot g(n) for all n \ge n_0.
- Describes the worst-case growth rate.

Examples of Big-O:

- 5n+3 \in O(n) (Linear)
- 2n^2 + 100n \in O(n^2) (Quadratic)
- \log_2 n + 5 \in O(\log n) (Logarithmic)
- 2^n + n^{10} \in O(2^n) (Exponential)

Other Notations:

- **Big-Omega (\Omega(g(n))):** Lower Bound (best-case or minimum growth).
- **Big-Theta (\Theta(g(n))):** Tight Bound (average-case or exact growth when upper and lower bounds match).

Growth Rate Hierarchy (from slowest to fastest):

1. O(1) (Constant)
2. O(\log n) (Logarithmic)
3. O(\sqrt{n}) (Square Root)
4. O(n) (Linear)
5. O(n \log n) (Linearithmic)
6. O(n^k) for k > 1 (Polynomial, e.g., O(n^2), O(n^3))
7. O(k^n) for k > 1 (Exponential, e.g., O(2^n))
8. O(n!) (Factorial)

Visualizing Growth:

- **Constant:** Flat line.
- **Logarithmic:** Grows very slowly.
- **Linear:** Straight line with positive slope.
- **Polynomial:** Curves upwards increasingly steeply.
- **Exponential/Factorial:** Explodes rapidly.

Rules for Big-O:

- **Constants:** Ignore constant factors. O(c \cdot f(n)) = O(f(n))
- **Sums:** Keep the dominant term. O(f(n) + g(n)) = O(\max(f(n), g(n)))
- **Products:** Multiply the complexities. O(f(n) \cdot g(n)) = O(f(n)) \cdot O(g(n))
- **Polynomials:** The highest degree term dominates. O(n^k + n^j) = O(n^k) if k>j

Example (Rule Application):

- O(3n^2 + 5n \log n + 100) = O(3n^2) = O(n^2)

Practical Implications: An algorithm with O(n) complexity is generally preferred over O(n^2) for large inputs, and O(n^2) over O(2^n).

Key Insight: Asymptotic notation provides a high-level, language-independent way to compare the efficiency of algorithms, focusing on how their performance scales with increasing input size.