**What is Time Complexity in Computer Science**

Time complexity describes the computational complexity that quantifies the amount of time an algorithm takes to run as a function of the length of the input. It's commonly expressed using Big O notation.

**Common Time Complexities:**

* **O(1)**: Constant time (e.g., array access)
* **O(log n)**: Logarithmic time (e.g., binary search)
* **O(n)**: Linear time (e.g., linear search)
* **O(n log n)**: Linearithmic time (e.g., efficient sorting algorithms)
* **O(n²)**: Quadratic time (e.g., bubble sort)
* **O(2ⁿ)**: Exponential time (e.g., brute-force algorithms)

**Search Algorithm Types**

**1 Linear Search**

* **Methodology**: Sequentially checks each element
* **Time Complexity**:
  + Best: O(1)
  + Average: O(n)
  + Worst: O(n)
* **Use Case**: Unsorted or small datasets

**2.2 Binary Search**

* **Methodology**: Divide-and-conquer approach
* **Requirements**: Sorted array
* **Time Complexity**:
  + Best: O(1)
  + Average/Worst: O(log n)

**2.3 Hash Table Search**

* **Methodology**: Uses hash functions for direct access
* **Time Complexity**:
  + Best/Average: O(1)
  + Worst: O(n)
* **Components**:
  + Hash function
  + Collision resolution method
  + Storage array

**3. Collision Resolution Techniques**

**3.1 Separate Chaining**

* **Concept**: Uses linked lists at each bucket
* **Advantages**:
  + Simple implementation
  + Handles arbitrary number of collisions
* **Disadvantages**:
  + Extra memory for pointers
  + Poor cache performance

**3.2 Open Addressing**

* **Variations**:
  + Linear probing
  + Quadratic probing
  + Double hashing
* **Load Factor**: Should be kept below 0.7
* **Advantages**:
  + Better cache performance
  + No extra memory allocation
* **Disadvantages**:
  + More complex deletion
  + Performance degrades as table fills

**3.3 Robin Hood Hashing**

* **Key Idea**: Balances probe sequence lengths
* **Benefits**: More stable performance
* **Drawbacks**: Complex implementation

**3.4 Cuckoo Hashing**

* **Method**: Uses two hash tables
* **Characteristics**:
  + Constant worst-case lookup time
  + May need rehashing
* **Insertion Process**:
  1. Insert into primary location
  2. If occupied, evict existing item to its alternate location
  3. Repeat until success or max iterations reached

**4. Comparative Analysis**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Space Complexity** | **Ideal Use Case** |
| --- | --- | --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) | O(1) | Small/unsorted datasets |
| Binary Search | O(1) | O(log n) | O(log n) | O(1) | Large sorted datasets |
| Hash Table | O(1) | O(1) | O(n) | O(n) | Frequent lookups |
| Jump Search | O(1) | O(√n) | O(√n) | O(1) | Ordered arrays |
| Interpolation | O(1) | O(log log n) | O(n) | O(1) | Uniformly distributed data |
| Exponential | O(1) | O(log n) | O(log n) | O(1) | Unlimited/infinite datasets |

**5. Practical Recommendations**

1. **For small datasets (<100 items)**: Linear search is often sufficient
2. **For sorted data**: Binary search provides optimal performance
3. **For frequent lookups**: Hash tables with good collision resolution
4. **For memory-constrained systems**: Consider open addressing
5. **For stable performance**: Robin Hood hashing offers good balance

**6. Advanced Topics (Brief)**

* **Bloom Filters**: Space-efficient probabilistic data structure
* **Trie Structures**: For prefix-based searching
* **B-trees**: For disk-based searching