Efficient code flows,
Algorithms dance, precise,
Complexity tamed.

ADT Recap

(ADTs are *menus, not dinners)*

List *(alias Vector)*

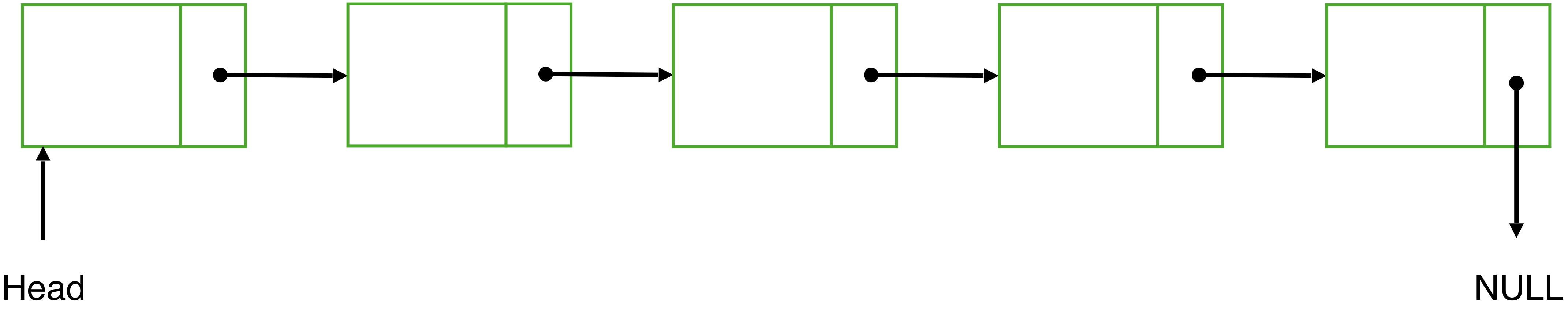
```
public interface List<E> {
    void add(int index, E element);
    E get(int index);
    E remove(int index);
    int size();
    boolean isEmpty();
}
```

```
Python: […,…,…]
```

Set *(alias Bag)*

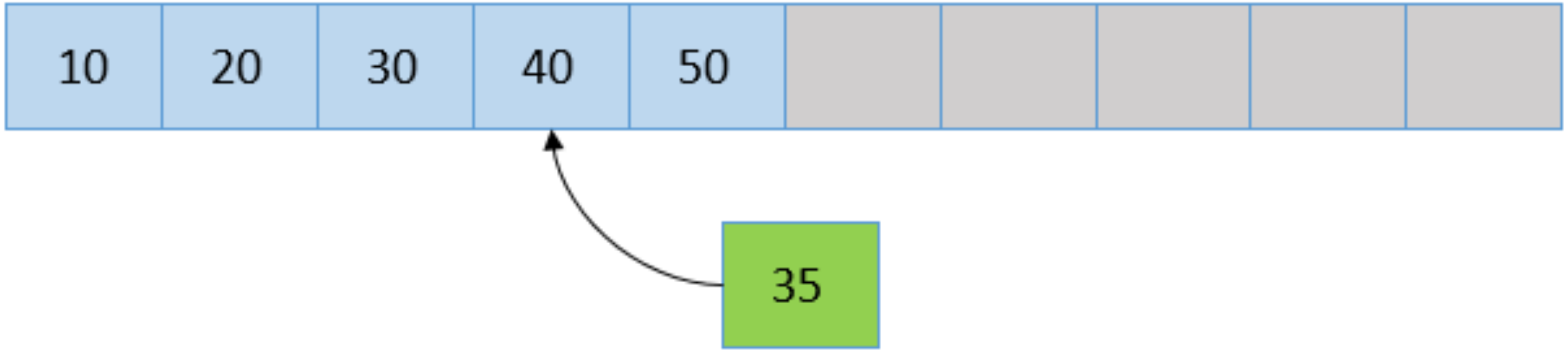
```
public interface Set<E> {
    boolean add(E element);
    boolean contains(E element);
    boolean remove(E element);
    int size();
    boolean isEmpty();
}
```

```
Python: {…,…,…}
```




```java
public void insertFront(E element) {
    Node<E> newNode = new Node<E>(element, firstNode);
    firstNode = newNode;
    if (size == 0) { // update lastNode if the list was empty
        lastNode = newNode;
    }
    size = size + 1;
}
```

```
nd) {
 indexToFind, we return the element

oFind);
```




```java
public void insertFront(E element) {
    increaseArrayDimensionIfFull();

    // shift all the elements from the last to the first one position right
    for (int i = size; i > 0; i--) {
        elements[i] = elements[i - 1];
    }

    elements[0] = element;
    size = size + 1;
}
```

Find Max

6	14	7	4	9	3	1	3

Find 11

1	3	4	6	8	11	12	15	30

11	12	15	30

11	12

With each comparison, the interval size is halved.

1. After the 2nd comparison, **n/4** elements remain, then **n/8**, etc.
2. The interval size becomes $\mathbf{n/2^k}$
3. In the worst case we continue until $\mathbf{n/2^k = 1}$
4. Solving $\mathbf{n/2^k = 1}$ for k gives us $\mathbf{k = \log_2(n)}$

Hence binary search is ***O(log n)***

O

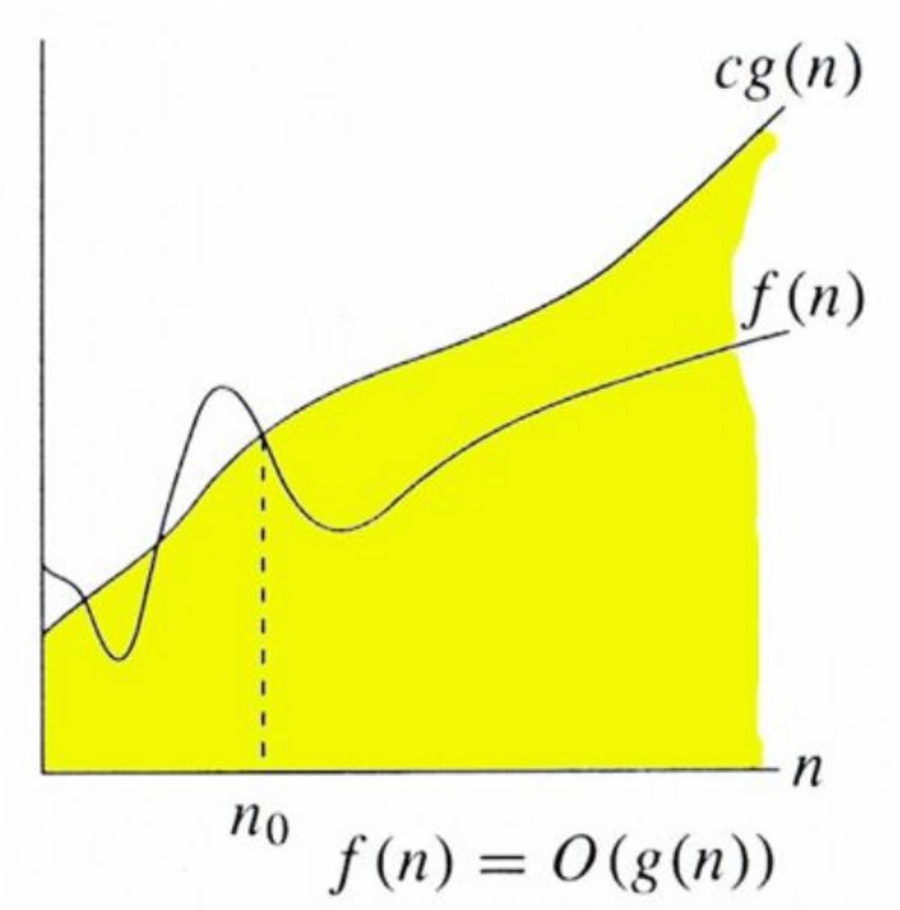
$O(g(n)) = \{$

$f(n)$: $\exists$ c and $n0$, s.t.

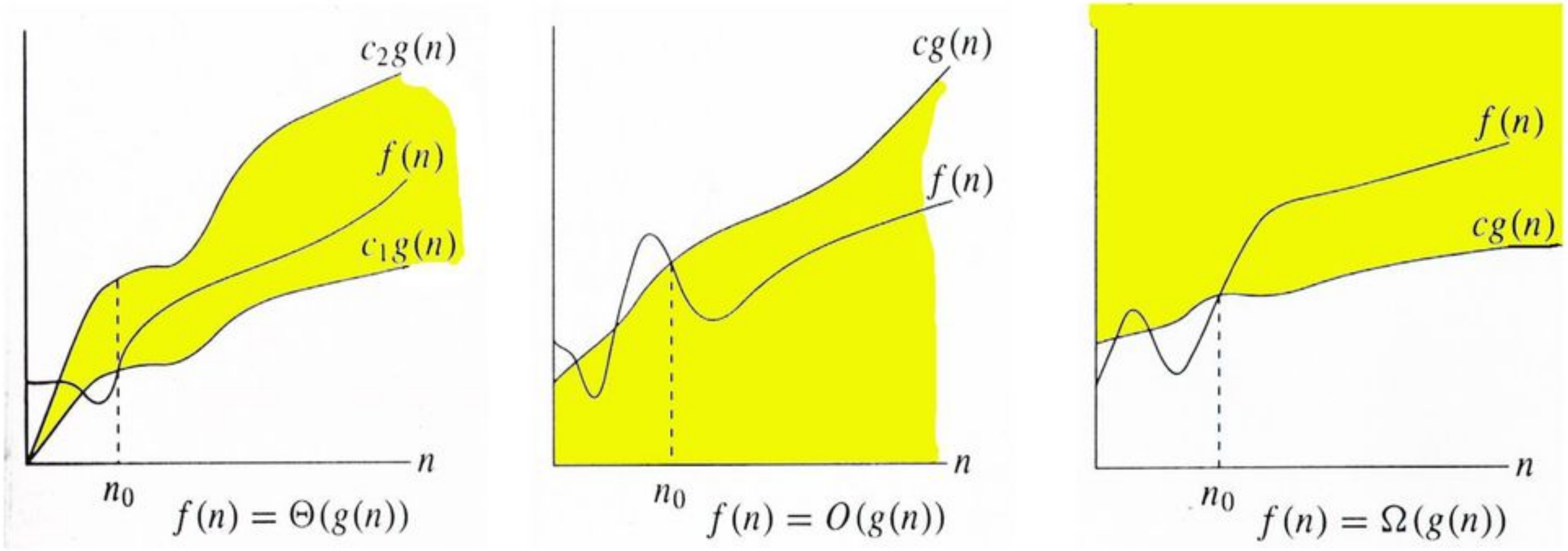
$0 \leq f(n) \leq cg(n)$

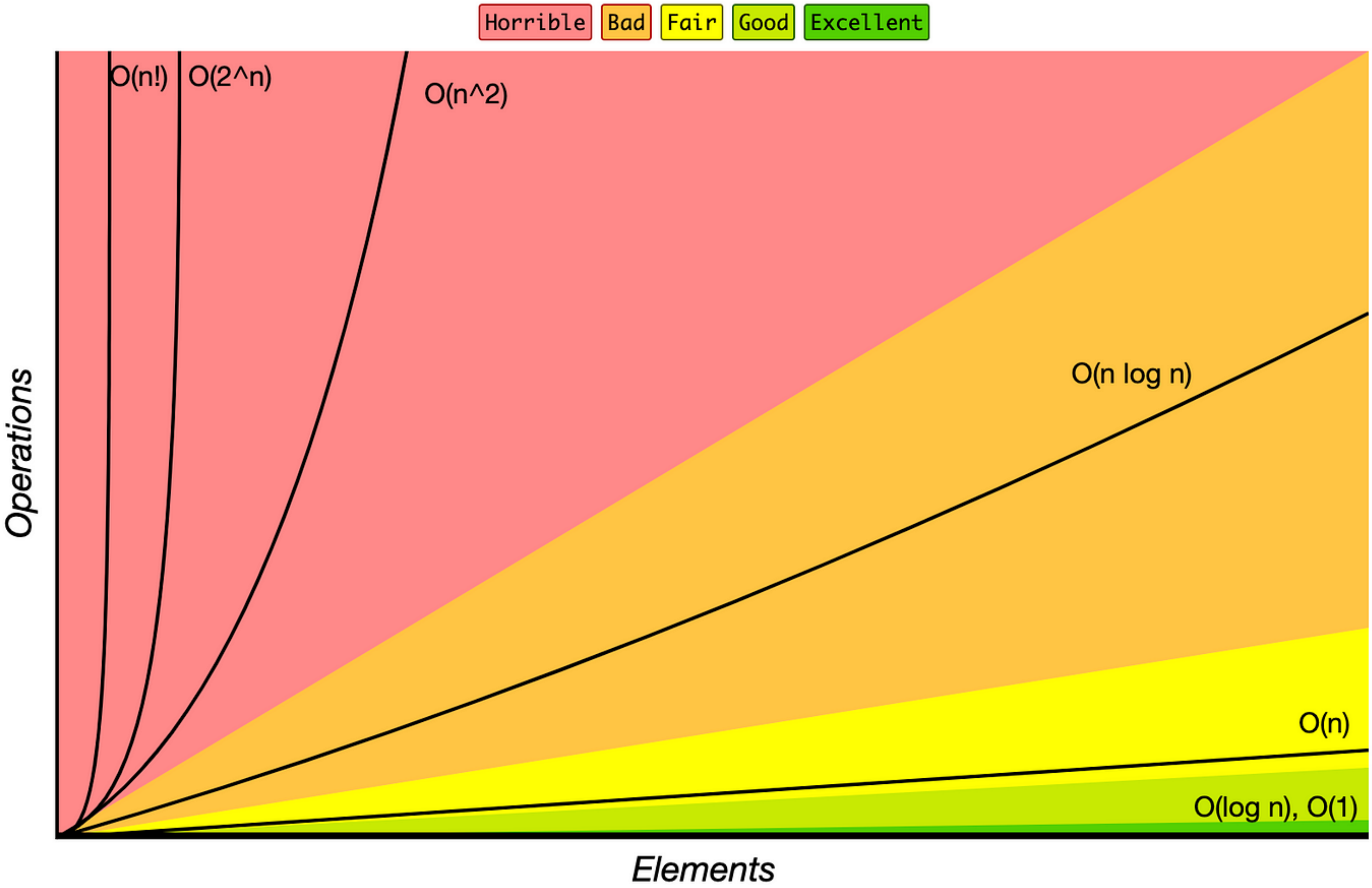
for all $n \geq n0$

$\}$



Θ / O / Ω


$c_2 g(n)$
$f(n)$
$c_1 g(n)$
n
n_0
$f(n) = \Theta(g(n))$
$cg(n)$
$f(n)$
n
n_0
$f(n) = O(g(n))$
$f(n)$
$cg(n)$
n
n_0
$f(n) = \Omega(g(n))$


Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

Common Complexity Classes

Class	Name	Examples
`O(1)`	Constant	Basic commands, Getting a value from an Array, adding an element to the front of a linked list.
`O(log n)`	Logarithmic	Typically seen in algorithms that break the problem in half every iteration. Such as Binary Search, finding an element in a balanced binary search tree.
`O(n)`	Linear	Searching for an element in an unsorted array, getting a value from a linked-list
`O(n log n)`	Log Linear	Efficient Sorting
$O(n^2)$	Quadratic	Bubble sort, Nested iterations such as checking for duplicates
$O(2^n)$	Exponential	Recursive Branching problems, e.g. naive Fibonacci, naïve Travelling Salesman

	Access	Search	Insertion	Deletion	Remarks
Linked List	O(n)	O(n)	O(1)*	O(1)*	* Assuming you have a reference to the insert/delete position (otherwise same as search)
Array List	O(1)	O(n)	O(n)*	O(n)	* *Amortized* O(1) for insertion at the end
Naive Unordered Set	-	O(n)	O(n)*	O(n)	Implemented as a simple array; checks for uniqueness on insert * *Amortized* O(1) for insertion at the end
Naive Ordered Set	-	O(log n)	O(n)	O(n)	Implemented as a sorted array
Linear Search	-	O(n)	-	-	Applicable to unsorted data; straightforward check of each element
Binary Search	-	O(log n)	-	-	Requires sorted data; not applicable for insertion/deletion without additional context

```
int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        sum++;
    }
}
```

1

n

n

$1 + 2 + 3 + 4 + \ldots + n$

$n(n\cancel{+1})$

$\cancel{2}$

$O(n^2)$

Sum of the First n Positive Integers

Let $S_n = 1 + 2 + 3 + 4 + \cdots + n = \sum_{k=1}^{n} k$. The elementary trick for solving this equation (which Gauss is supposed to have used as a child) is a rearrangement of the sum as follows:

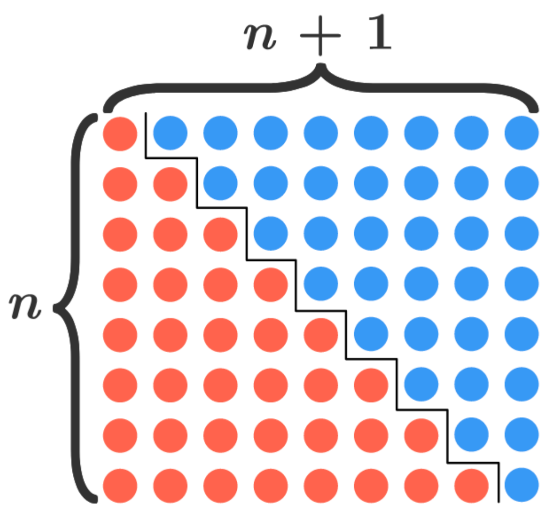
$$\begin{aligned} S_n &= 1 + \quad 2 + \quad 3 + \cdots + \quad n \\ S_n &= n + \quad n-1 + \quad n-2 + \cdots + \quad 1. \end{aligned}$$

Grouping and adding the above two sums gives

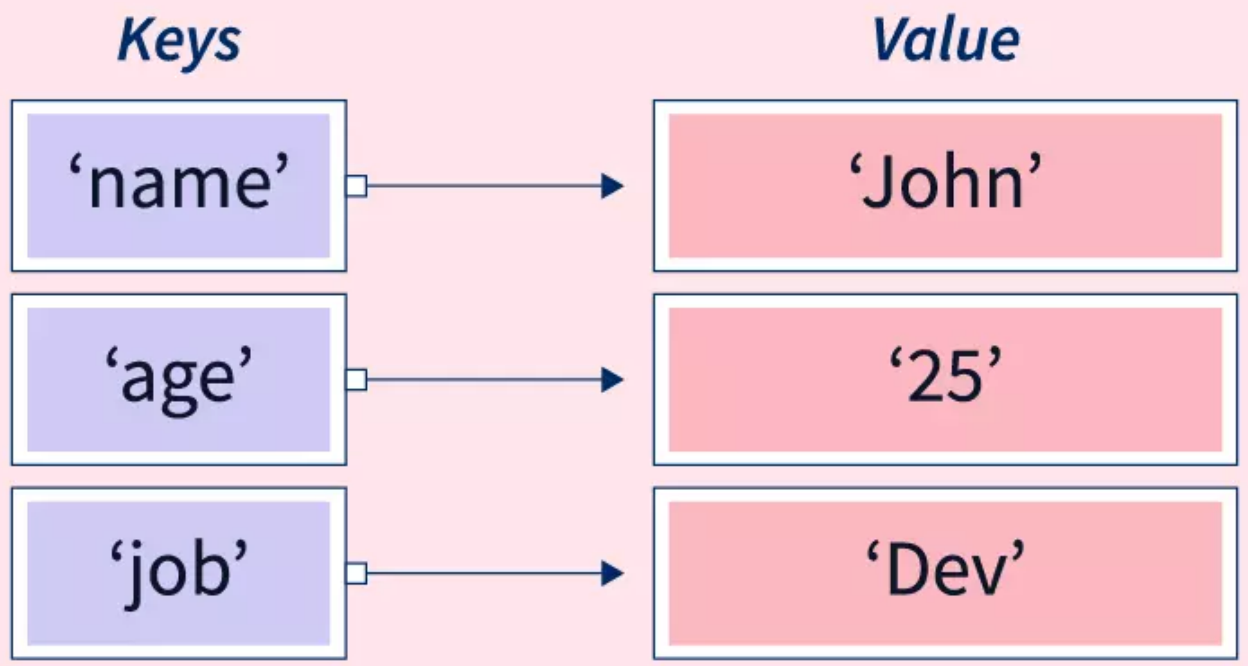
$$\begin{aligned} 2S_n &= (1+n) + (2+n-1) + (3+n-2) + \cdots + (n+1) \\ &= \underbrace{(n+1) + (n+1) + (n+1) + \cdots + (n+1)}_{n \text{ times}} \\ &= n(n+1). \end{aligned}$$

Therefore,

$$S_n = \frac{n(n+1)}{2}.$$



Words in a map's world,
Collisions echo silence,
Dictionary's realm.


Keys
Value
‘name’
‘John’
‘age’
‘25’
‘job’
‘Dev’

Dictionary ADT *(alias Map/Table)*

```
public interface Dictionary<K, V> {
        // Associates the specified value with the specified key
        void put(K key, V value);
        // Returns the value to which the specified key is mapped
        V get(K key);
        // Removes the mapping for a key if it is present
        V remove(K key);
        // Checks if the dictionary contains a mapping for the specified key
        boolean containsKey(K key);
        // Returns the number of key-value mappings in the dictionary
        int size();
        // Checks if the dictionary is empty
        boolean isEmpty();
}
```

`Python: {key:value, …}`

HashMap

- Provide efficient data retrieval, insertion, and deletion operations by mapping **keys** to **values** using a **hash function**, achieving expected-case time complexity of ***O(1)*** for these operations.
- It optimizes data access by minimizing the need for sequential search through keys, making it ideal for scenarios where quick lookup of information is critical.

Which data structure gives us ***O(1)*** access and what are its limits?

What is a Key?

How large can my keys get? How can they fit my array?

key-space **>** *capacity*

Hashing & Collision

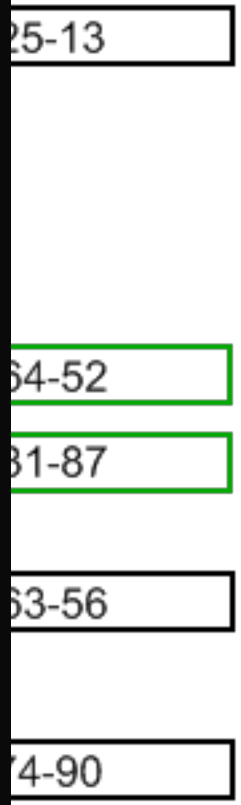
mod (%) operator – remainder of division
Result: constraining numbers within a specific range

$h(x) = x \% N$

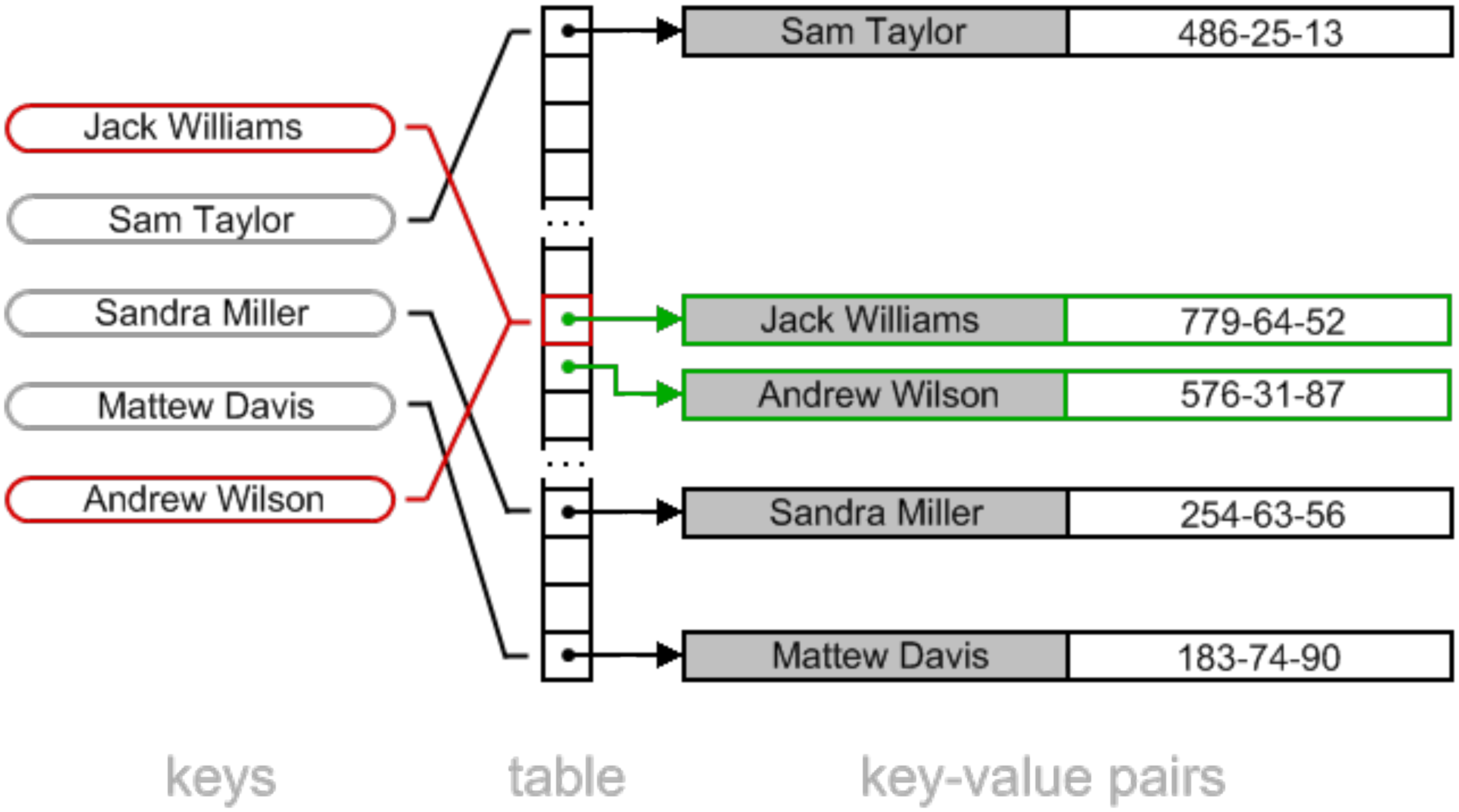
$N = 8$

Separate Chaining

```
class HashNode<K, V> {
    K key;
    V value;
    HashNode<K, V> next;

    public HashNode(K key, V value) {
        this.key = key;
        this.value = value;
        this.next = null;
    }
}

class SimpleHashMap<K, V> {
    private HashNode<K, V>[] chainArray;
    private int capacity; // Size of the array
    private int size; // Number of key-value pairs in the HashMap

    public SimpleHashMap(int capacity) {
        this.capacity = capacity;
        this.chainArray = new HashNode[capacity];
        this.size = 0;
    }

    private int hashFunction(K key) {
        return Math.abs(key.hashCode()) % capacity;
    }
```



Open Addressing (Linear Probing)



Operation	Expected Case	Worst Case	Remarks
Insert	O(1)	O(n)	Worst case occurs when all keys hash to the same bucket.
Delete	O(1)	O(n)	Similar to insert, depends on the number of items in a bucket.
Search	O(1)	O(n)	Worst case occurs when searching for a non-existent key that hashes to a heavily populated bucket.