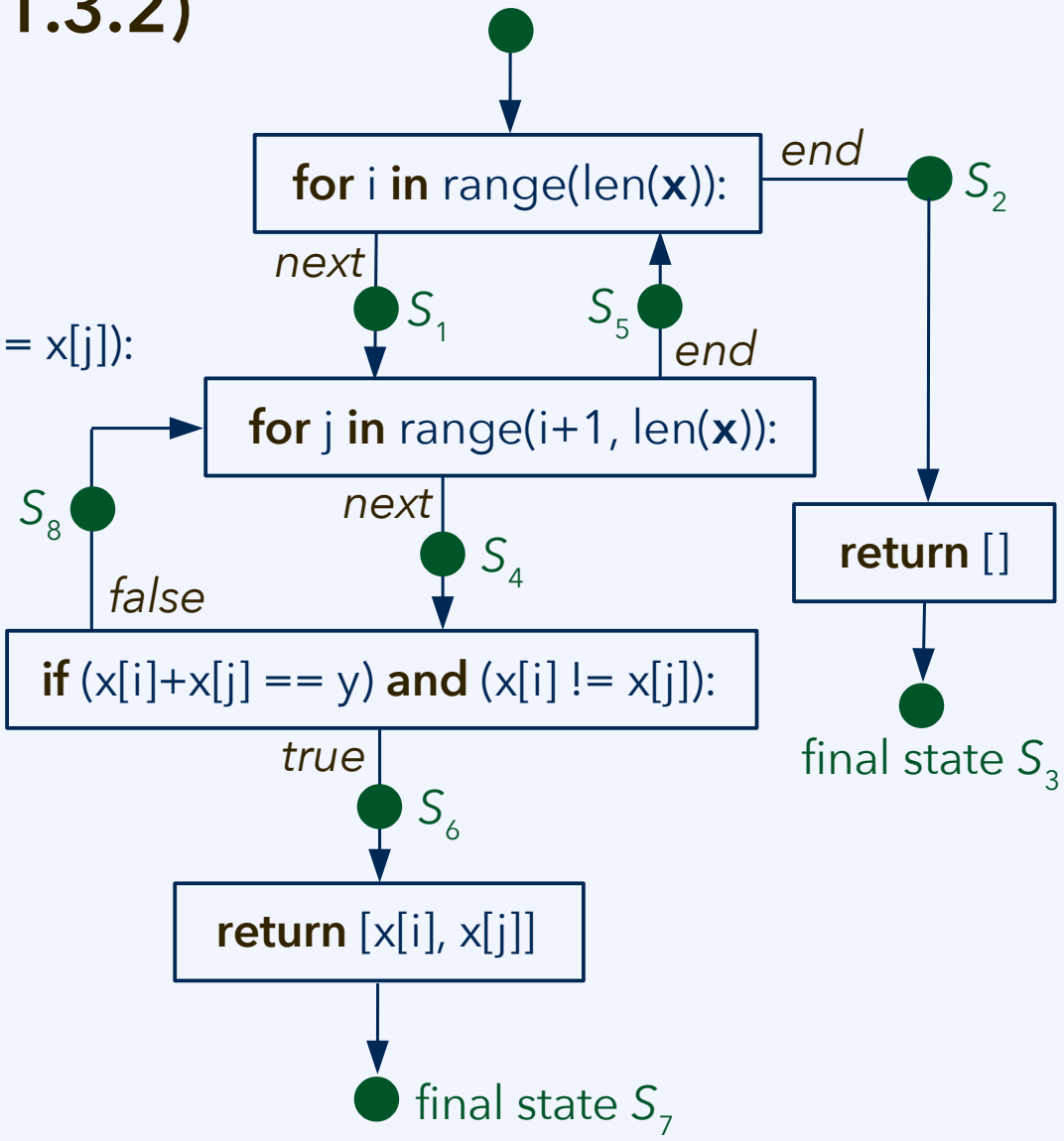
# Number matching (T1.3.2)

```
def natmatch(x, y):
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if (x[i]+x[j] == y) and (x[i] != x[j]):
                return [x[i], x[j]]
    return []
```

**Specification**

The function takes a list **x** and a natural number *y* as arguments. If in the list **x**, there are elements a and b which are not equal and add up to y, the list [a, b] is returned; otherwise, [] is returned.



# Number matching (T1.3.2)

```
def natmatch(x, y):
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if (x[i]+x[j] == y) and (x[i] != x[j]):
                return [x[i], x[j]]
    return []
```

**Note: Input size *n* given by len(x)**

loop executed O(*n*) times:

- loop executed O(*n*) times:
  - O(1) instructions
  - O(1) optional instructions

O(1) optional instructions

---

$O(n) \cdot O(n \cdot 1) + O(1) = O(n^2)$ instructions

---

**$O(n^2)$ time efficiency**

# Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan:

```
def natmatch(x, y):
    mydict = {}
    for i in range(len(x)):
        c = y - x[i]
        if c in mydict:
            return [c, x[i]]
        mydict[x[i]] = i
    return []
```

Python dictionaries and sets could be used to this effect equivalently.

Example, **x** = [6, 4, 5, 3, 9], y = 11:

6 → 11 – 6 = 5 not found in storage
insert 6 into storage

4 → 11 – 4 = 7 not found in storage
insert 4 into storage

5 → 11 – 5 = 6 found in storage
return [6, 5]

# Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan:

```
def natmatch(x, y):
    mydict = {}
    for i in range(len(x)):
        c = y - x[i]
        if c in mydict:
            return [c, x[i]]
        mydict[x[i]] = i
    return []
```

Python dictionaries and sets are implemented as dynamically resized **hash tables**:

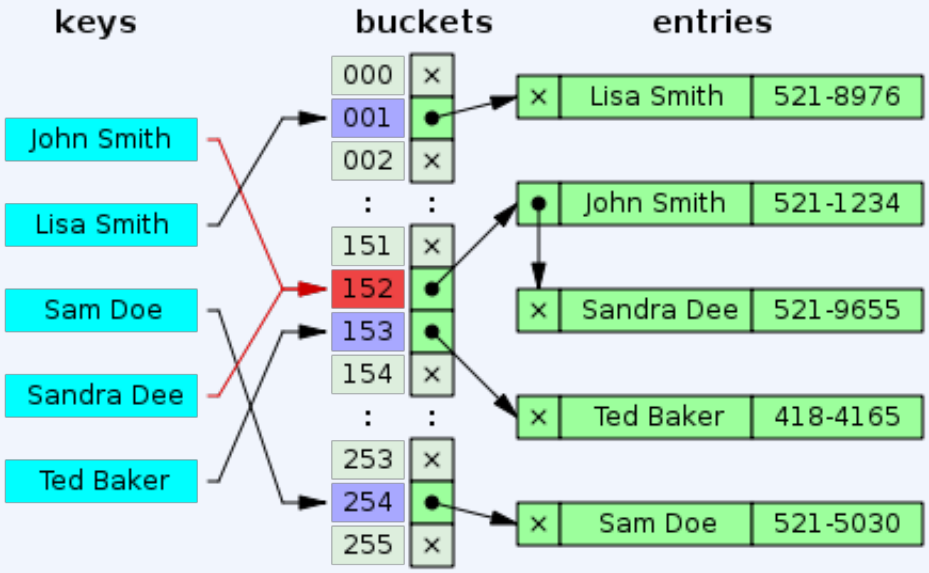



*Fig. from Wikipedia, "Hash table"*

# Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan; worst case still $O(n^2)$:

```
def natmatch(x, y):
    mydict = {}
    for i in range(len(x)):
        c = y - x[i]
        if c in mydict:
            return [c, x[i]]
        mydict[x[i]] = i
    return []
```

In the worst case, this data structure has $O(n)$ time for search and insertion. For the average case, it is highly efficient.

Python dictionaries and sets are implemented as dynamically resized **hash tables**:

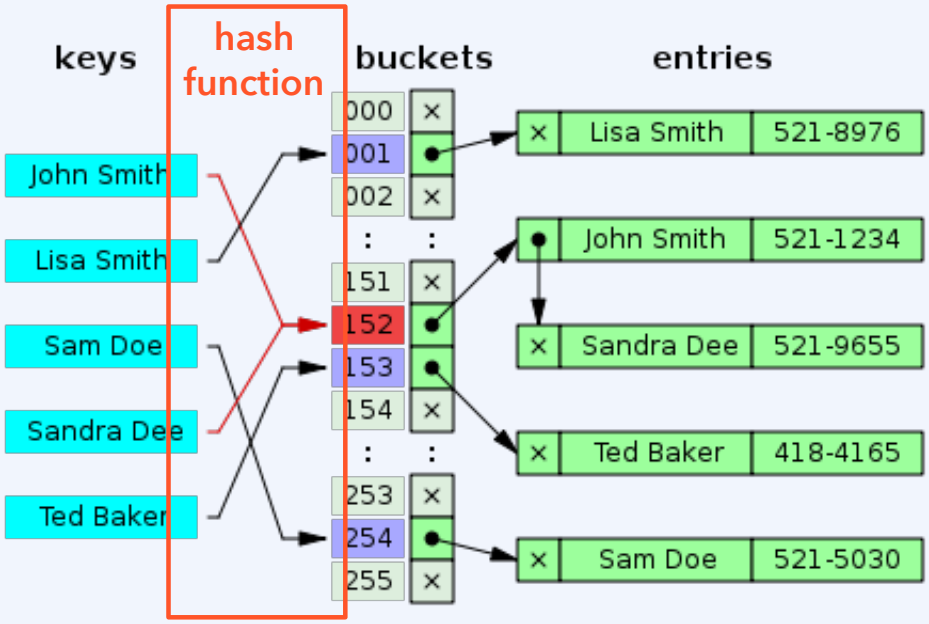



*Fig. from Wikipedia, "Hash table"*

# Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan:

```
def natmatch(x, y):
    initialize empty storage
    for i in range(len(x)):
        c = y - x[i]
        if storage.contains(c):
            return [c, x[i]]
        storage.insert(x[i])
    return []
```

O(*n*) loop operations.

Each with one **search** operation and one **insertion** operation.

What is the time efficiency? How does it depend on the employed data structure?

Example, **x** = [6, 4, 5, 3, 9], y = 11:

6 → 11 – 6 = 5 not found in storage
insert 6 into storage

4 → 11 – 4 = 7 not found in storage
insert 4 into storage

5 → 11 – 5 = 6 found in storage
return [6, 5]

# Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan:

```
def natmatch(x, y):
    initialize empty storage
    for i in range(len(x)):
        c = y - x[i]
        if storage.contains(c):
            return [c, x[i]]
        storage.insert(x[i])
    return []
```

O(*n*) loop operations.

Each with one **search** operation and one **insertion** operation.

What is the time efficiency? How does it depend on the employed data structure?

How about:

- Unsorted linked list or dyn. array?
- A sorted dynamic array?
- A sorted linked list or an unbalanced search tree?
- A balanced search tree?
- Python sets or dicts?

# Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan:

```
def natmatch(x, y):
    initialize empty storage
    for i in range(len(x)):
        c = y - x[i]
        if storage.contains(c):
            return [c, x[i]]
        storage.insert(x[i])
    return []
```

O(*n*) loop operations.

Each with one **search (s)** operation and one **insertion (i)** operation.

What is the time efficiency? How does it depend on the employed data structure?

How about:

- Unsorted linked list or dyn. array?
  **s** done in O(*n*), **i** done in O(1).
- A sorted dynamic array?
  **s** done in O(log *n*), **i** done in O(*n*).
- A sorted linked list or an unbalanced search tree?
  **s** done in O(*n*), **i** done in O(*n*).
- A balanced search tree?
  **s** and **i** both done in O(log *n*).
- Python sets or dicts?
  Worst case O(*n*) for both **s** and **i**.

# Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan:

```
def natmatch(x, y):
    initialize empty storage
    for i in range(len(x)):
        c = y - x[i]
        if storage.contains(c):
            return [c, x[i]]
        storage.insert(x[i])
    return []
```

O(*n*) loop operations.

- O(log *n*) time per iteration.

O(*n* log *n*) with a balanced tree.

What is the time efficiency? How does it depend on the employed data structure?

How about:

- Unsorted linked list or dyn. array?
  **s** done in O(*n*), **i** done in O(1).
- A sorted dynamic array?
  **s** done in O(log *n*), **i** done in O(*n*).
- A sorted linked list or an unbalanced search tree?
  **s** done in O(*n*), **i** done in O(*n*).
- A balanced search tree?
  **s** and **i** both done in O(log *n*).
- Python sets or dicts?
  Worst case O(*n*) for both **s** and **i**.