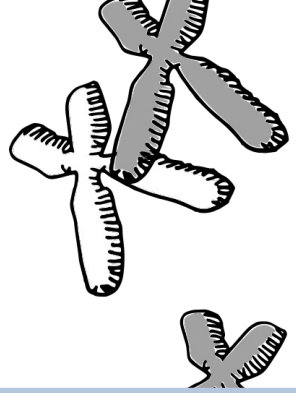
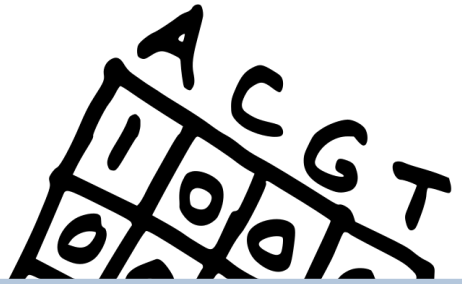
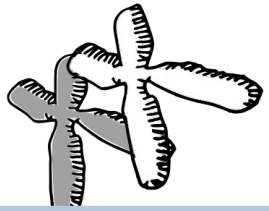




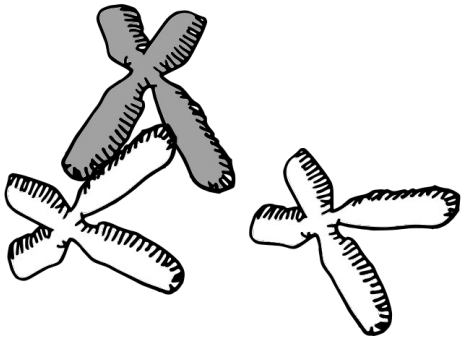


Sequence algorithms

DAG workshop, 2020

Cyril Matthey-Doret

$O(n \log n)$

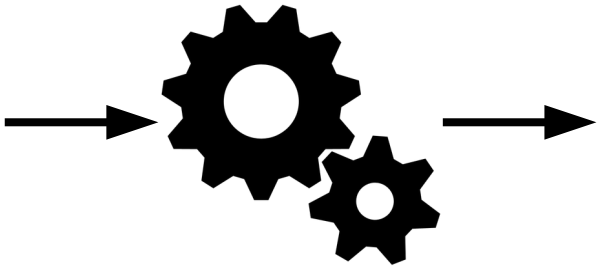


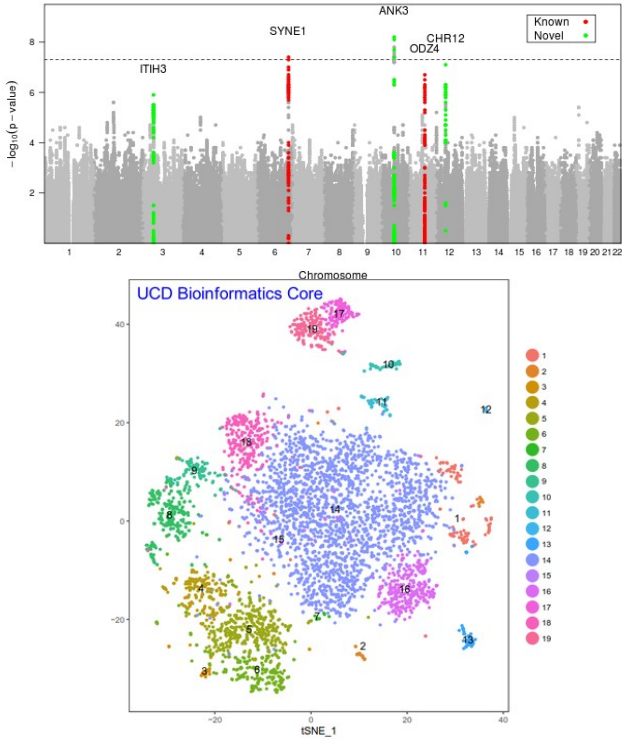
cyril.mattheydoret@gmail.com

cmdoret

Genomics

- Use DNA sequences to answer biological questions
- But how ?



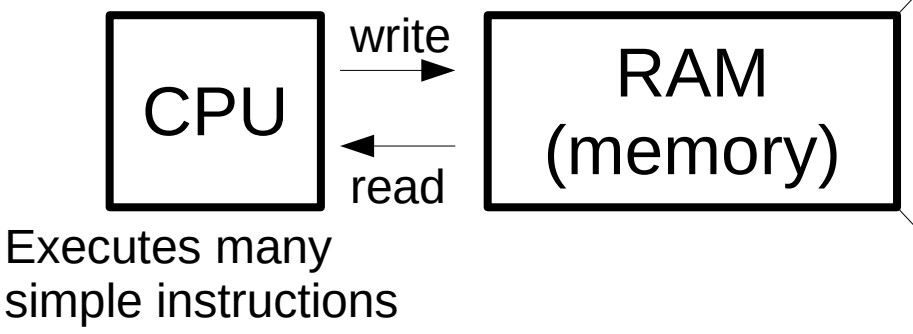


The tools

- Computer algorithms to automate tedious tasks
- We use programming languages to write algorithms

Thinking like a computer

- Computers can perform a limited set of instructions (i.e. comparisons, allocating / deallocating memory, ...)



address	content
100	instruction
101	data
102	instruction
103	empty
104	data

Example: max of an array

- Scripting languages make our lives easy

Python:

```
a = [1, 0, 31, 4]
max(a)
```

Example: max of an array

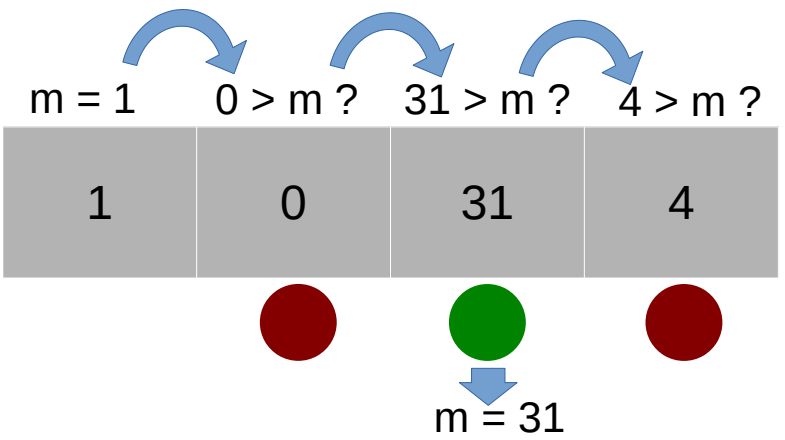
- Scripting languages make our lives easy

Python:

```
a = [1, 0, 31,
4]
max(a)
```

Algorithm:

```
a = [1, 0, 31, 4]
m = a[0]
for i in a[1:]:
    if i > m:
        m = i
```



Algorithms in genomics

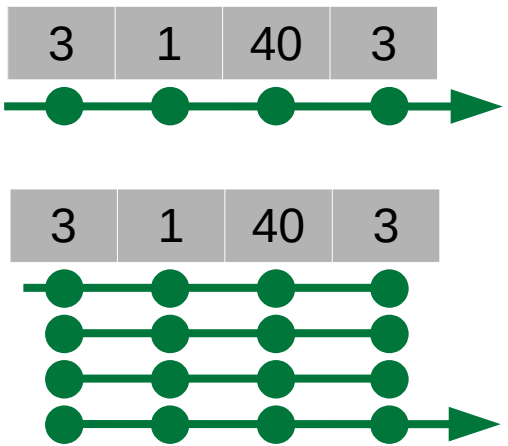
- Assembling genomes
- Aligning reads
- Cluster samples by gene expression similarity
- Predict gene functions from their sequence

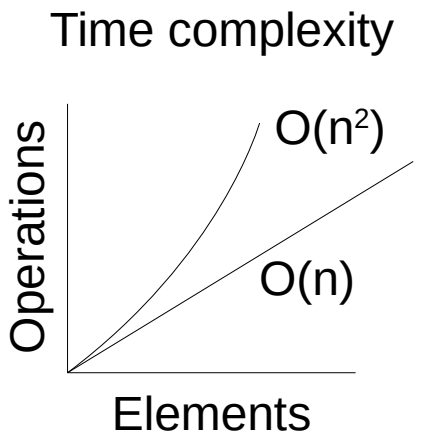
Time complexity

- Big O notation is used to describe run time

Examples:

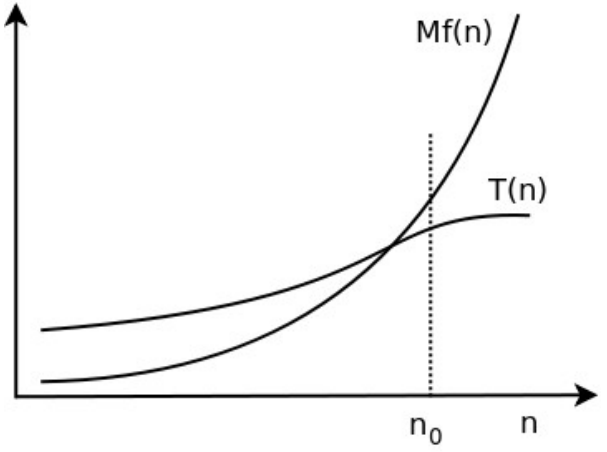
```
a = [3, 1, 40, 3]

for i in a:
    print(i)

for i in a:
    For i in a:
        print(i)
```





Big O notation: definition

- T(n) (your algorithm) has order of f(n) if there are positive constants M and n_0 such that $T(n) \leq M*f(n)$ for all $n \geq n_0$
- T(n) does not grow faster than f(n)



T(n) = O(f(n))

Big O notation: definition

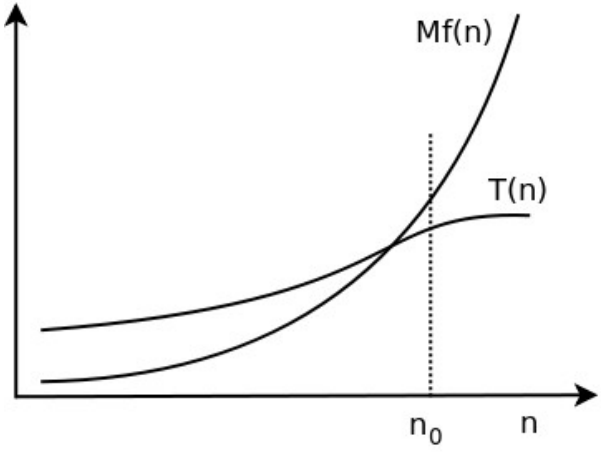
- T(n) (your algorithm) has order of f(n) if there are positive constants M and n_0 such that $T(n) \leq M*f(n)$ for all $n \geq n_0$
- T(n) does not grow faster than f(n)

Example:

If T(n) is O(1): $T(n) \leq M*1$

In english: If T(n) has order O(1), it will always be faster or same than a constant value M, no matter the input size

What if T(n) is O(n) ?



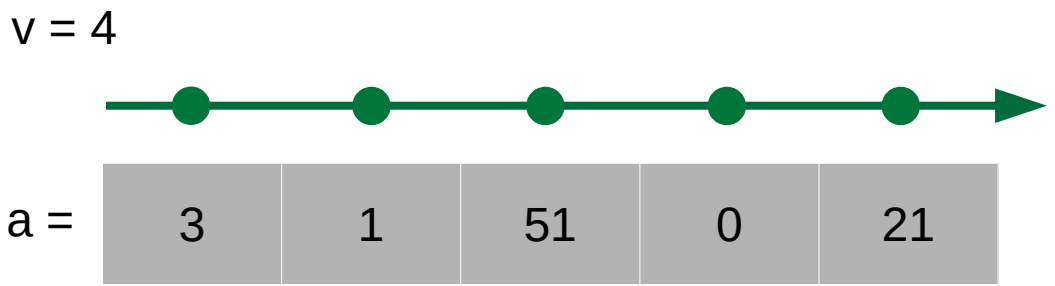
T(n) = O(f(n))

Example: Improving an algorithm

- Checking if value **v** is in an array **a**
- What is the time complexity ?

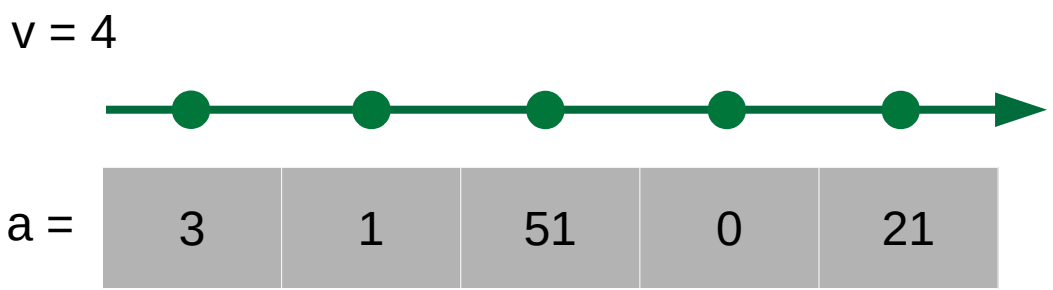
A naive approach

- Checking if value **v** is in an array **a**
- What is the time complexity ?



A naive approach

- Checking if value **v** is in an array **a**
- What is the time complexity ?



Naive implementation:

```
found = “no”
for i in a:
    if i == v:
        found = “yes”
```

O(n)

Is there a better way ?

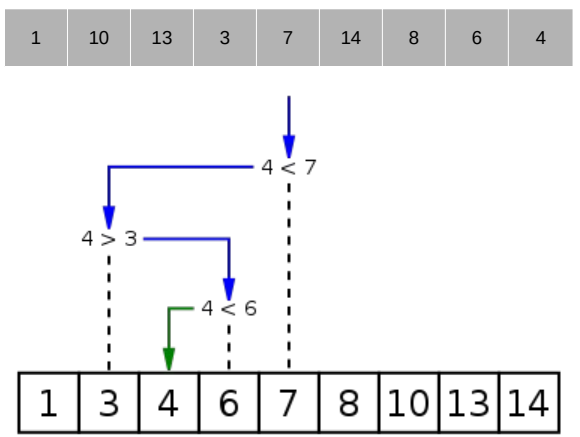
Getting faster

- Checking if value **v** is in an array **a**
- What is the time complexity ?

Sort array

Sorting costs **O(n log n)**

Binary search

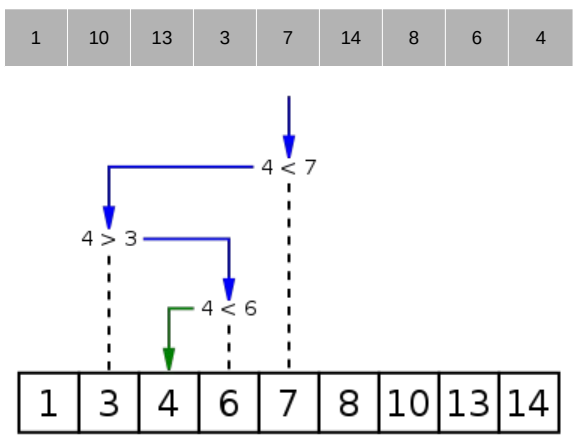


Getting faster

- Checking if value **v** is in an array **a**
- What is the time complexity ?

Sort array

Sorting costs **O(n log n)**



Binary search

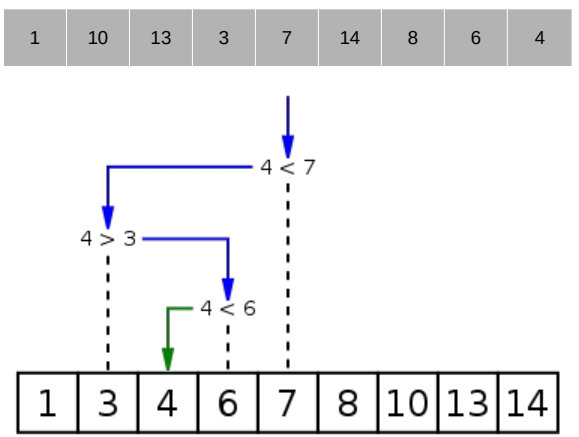
But allows to search in **O(log n)**
→ better than **O(n)**

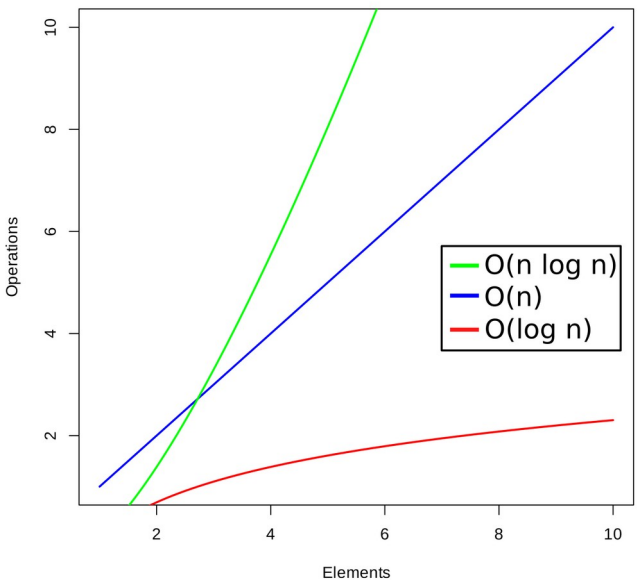
Getting faster

- Checking if value **v** is in an array **a**
- What is the time complexity ?

Sort array

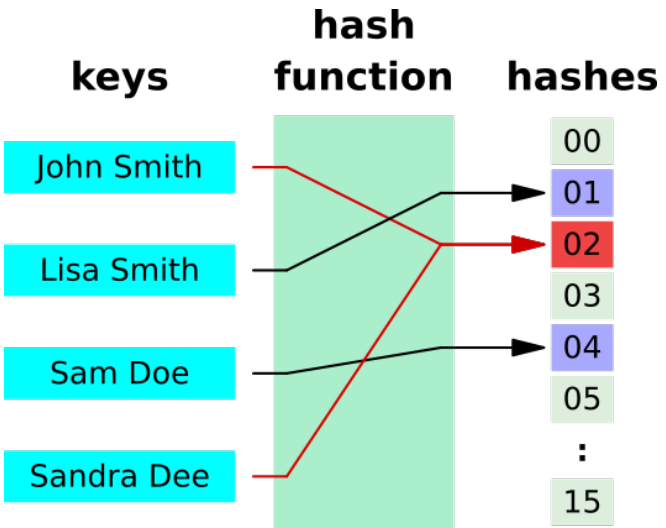
Binary search





Hashing to the rescue

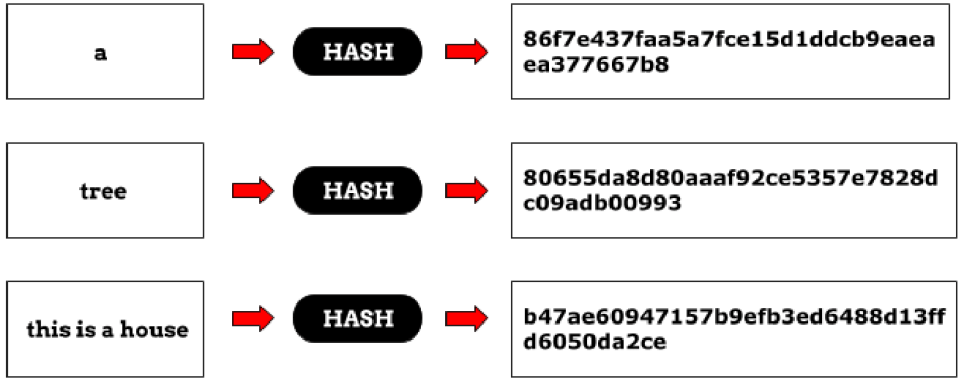
- Best solution: Hash table (aka maps, dicts)
- Data structures with a mapping function



Wait, what’s hashing ?

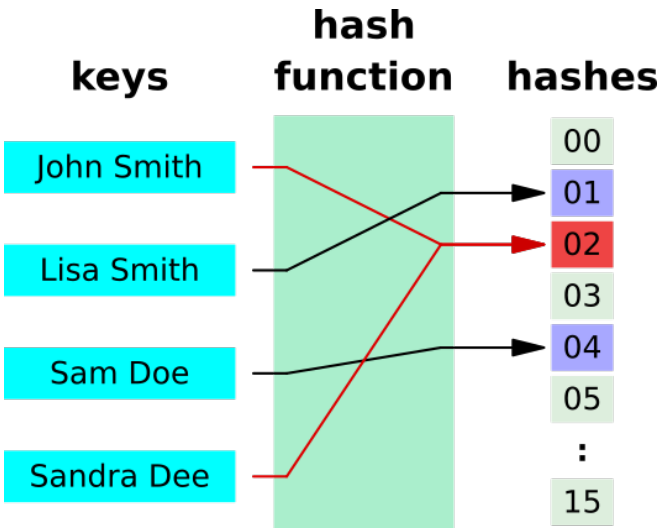
Hash function: “A function mapping data of arbitrary size to fixed-size values”

```
In [11]: v = 4
    ...: a = [13, 45, 10, 4]
    ...: hash_tbl = [[] for i in range(10)]
    ...:
    ...: # Dummy hash function
    ...: def hash_func(x):
    ...:     return x % 10
```



Hash tables

Hash function: “A function mapping data of arbitrary size to fixed-size values”

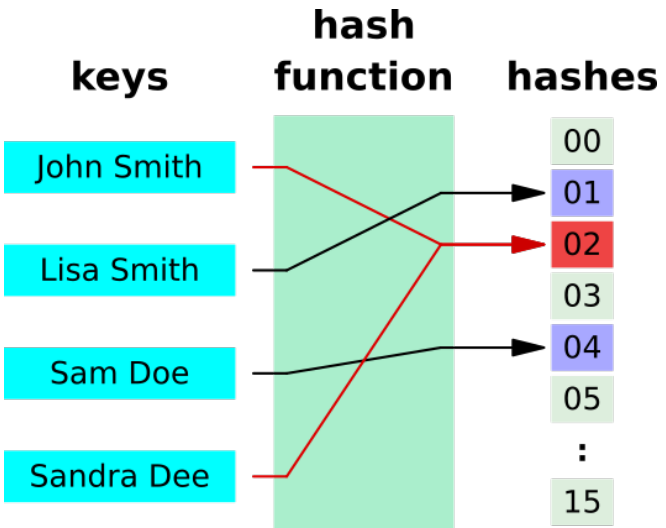


Fill the table

```
In [11]: v = 4
    ...: a = [13, 45, 10, 4]
    ...: hash_tbl = [[] for i in range(10)]
    ...:
    ...: # Dummy hash function
    ...: def hash_func(x):
    ...:     return x % 10
    ...:
    ...: for i in a:
    ...:     bucket = hash_func(i)
    ...:     hash_tbl[bucket].append(i)
```

Hash tables

Hash function: “A function mapping data of arbitrary size to fixed-size values”



Fill the table

```
In [11]: v = 4
    ...: a = [13, 45, 10, 4]
    ...: hash_tbl = [[] for i in range(10)]
    ...:
    ...: # Dummy hash function
    ...: def hash_func(x):
    ...:     return x % 10
    ...:
    ...: for i in a:
    ...:     bucket = hash_func(i)
    ...:     hash_tbl[bucket].append(i)
    ...:

In [12]: hash_tbl
Out[12]: [[10], [], [], [13], [4], [45], [], [], [], []]
```

Hash tables

Hash function: “A function mapping data of arbitrary size to fixed-size values”

Fill the table

Query the table

```
In [11]: v = 4
    ...: a = [13, 45, 10, 4]
    ...: hash_tbl = [[] for i in range(10)]
    ...:
    ...: # Dummy hash function
    ...: def hash_func(x):
    ...:     return x % 10
    ...:
    ...: for i in a:
    ...:     bucket = hash_func(i)
    ...:     hash_tbl[bucket].append(i)
    ...:

In [12]: hash_tbl
Out[12]: [[10], [], [], [13], [4], [45], [], [], [], []]

In [13]: # Is v in a ?

In [14]: v_bucket = hash_func(v)

In [15]: hash_tbl[v_bucket]
Out[15]: [4]

In [16]: # Yes
```

Hash tables

Hash function: “A function mapping data of arbitrary size to fixed-size values”

What’s the time complexity of filling and querying ?

Fill the table

Query the table

```
In [11]: v = 4
    ...: a = [13, 45, 10, 4]
    ...: hash_tbl = [[] for i in range(10)]
    ...:
    ...: # Dummy hash function
    ...: def hash_func(x):
    ...:     return x % 10
    ...:
    ...: for i in a:
    ...:     bucket = hash_func(i)
    ...:     hash_tbl[bucket].append(i)
    ...:

In [12]: hash_tbl
Out[12]: [[10], [], [], [13], [4], [45], [], [], [], []]

In [13]: # Is v in a ?

In [14]: v_bucket = hash_func(v)

In [15]: hash_tbl[v_bucket]
Out[15]: [4]

In [16]: # Yes
```

Hash tables

Hash function: “A function mapping data of arbitrary size to fixed-size values”

What’s the time complexity of filling and querying ?

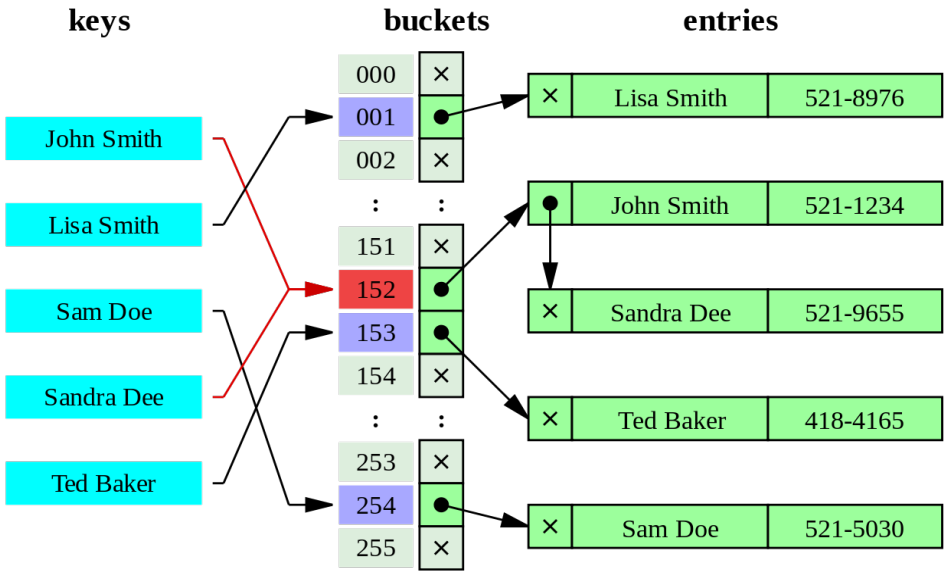
Fill the table
O(n)

Query the table
O(1)

```
In [11]: v = 4
    ...: a = [13, 45, 10, 4]
    ...: hash_tbl = [[] for i in range(10)]
    ...:
    ...: # Dummy hash function
    ...: def hash_func(x):
    ...:     return x % 10
    ...:
    ...: for i in a:
    ...:     bucket = hash_func(i)
    ...:     hash_tbl[bucket].append(i)
    ...:

In [12]: hash_tbl
Out[12]: [[10], [], [], [13], [4], [45], [], [], [], []]

In [13]: # Is v in a ?

In [14]: v_bucket = hash_func(v)

In [15]: hash_tbl[v_bucket]
Out[15]: [4]

In [16]: # Yes
```

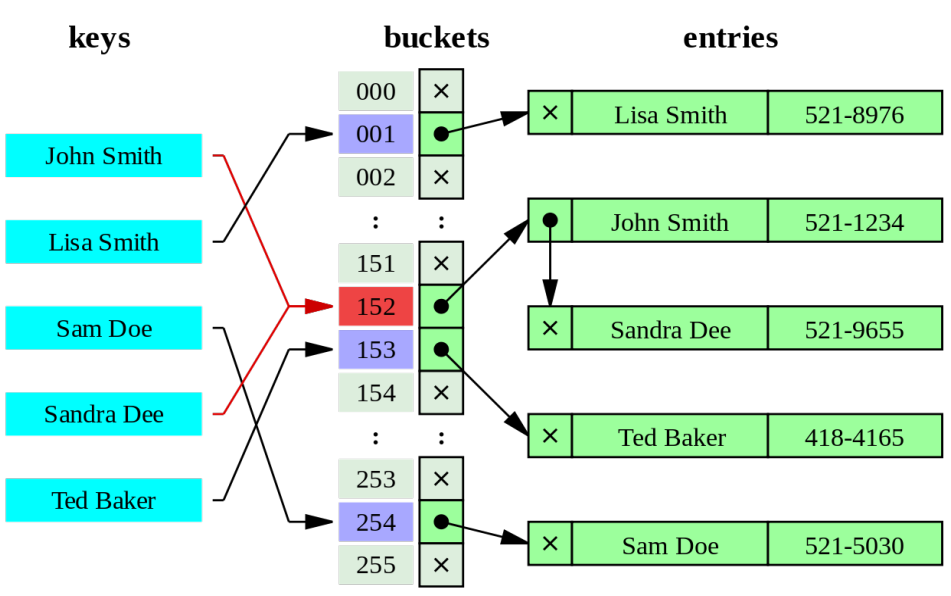
Hash collisions

- What happens when multiple hash have the same hash ?
- Multiple entries stored in the same bucket



Hash collisions

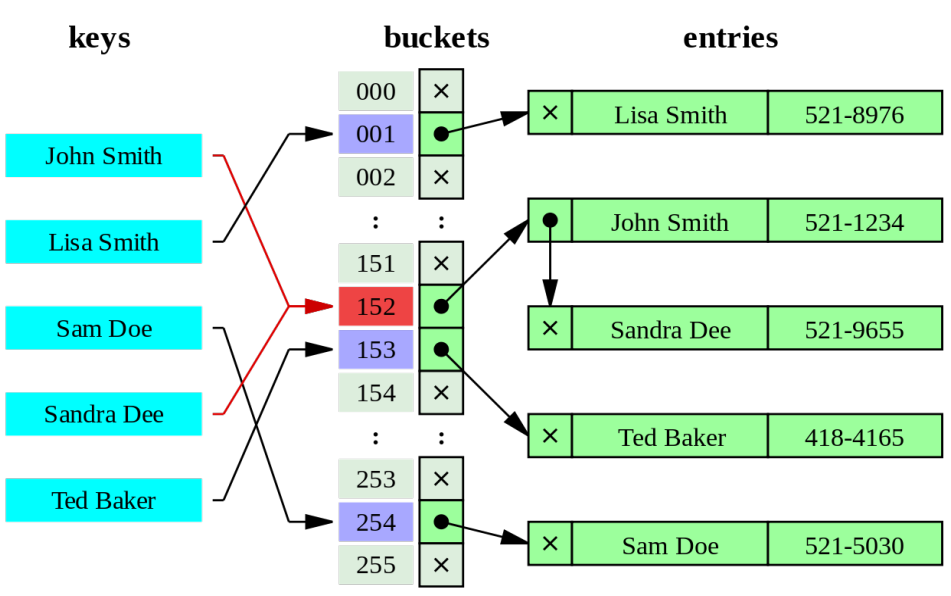
- What happens when multiple hash have the same hash ?
- Multiple entries stored in the same bucket




```
In [22]: hash_tbl
Out[22]: [[10], [], [], [13], [4], [45], [], [], [], []]

In [23]: hash_tbl[hash_func(23)].append(23)

In [24]: hash_tbl
Out[24]: [[10], [], [], [13, 23], [4], [45], [], [], [], []]
```

Hash collisions

- What happens when multiple hash have the same hash ?
- Multiple entries stored in the same bucket

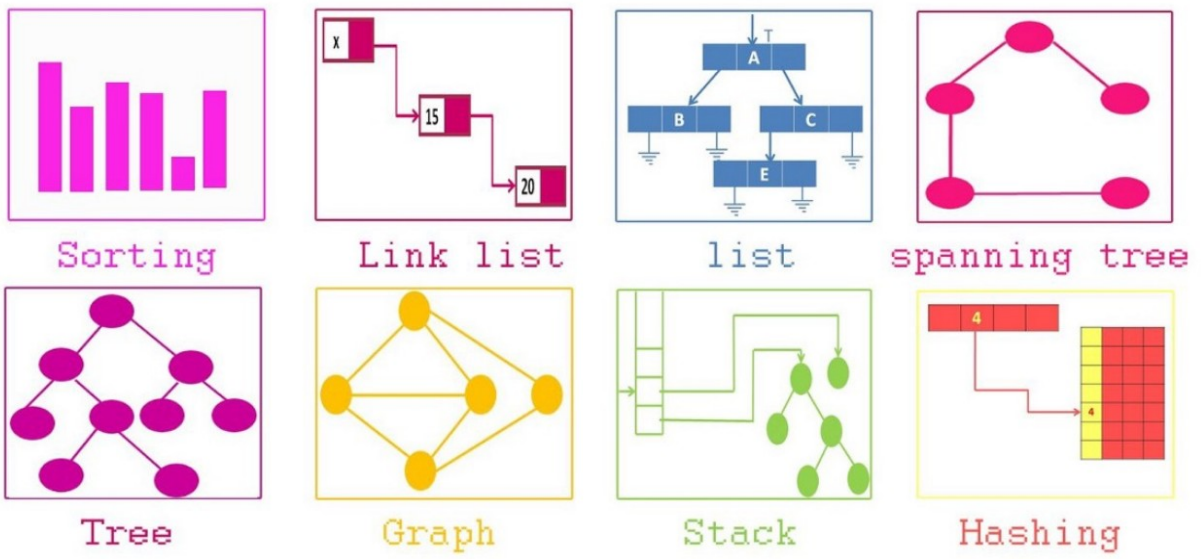



```
In [22]: hash_tbl
Out[22]: [[10], [], [], [13], [4], [45], [], [], [], []]

In [23]: hash_tbl[hash_func(23)].append(23)

In [24]: hash_tbl
Out[24]: [[10], [], [], [13, 23], [4], [45], [], [], [], []]
```

What does that imply for query time ?

Data structures as a tool

- Storing our data in a hash table enabled O(1) query time
- Data structures are as important as the algorithm



Exercises

- Provided as a jupyter notebook (requires python)
- Apply knowledge on biological data
- A bit harder :)