Scientific Computation

Spring 2019

Lecture 4

Imperial College
London

22 January, 2019

Today

- Constant-time search

- Lecture 2: Binary search applied to sorted array requires $\log_2 N$ operations
- Can we do better?

- **Lecture 2: Binary search applied to sorted array requires $\log_2 N$ operations**

- **Can we do better?**

- **Proposal: Set the value of an integer as index in an array**

6	8	23	32

  - **Then for the example above we would have an array (or list),** X, **where** X[6] = X[8] = X[23] = X[32] = 1

  - **Speed for search would be constant time** (O(1))

  - **But many weaknesses:**
    - **What about more general data (not just non-negative integers)**
    - **Inserting and deleting entries would require** O(N) **time**

Searching

- Lecture 1: Binary search applied to sorted array requires $\log_2 N$ operations
- Can we do better?
- Real-world problem:
  - A startup is keeping track of unique visitors to its website each month
  - Each visitor must be assessed as “new” or “repeat”
  - Visitors who have not visited in 30+ days must be removed
  - Everything must be fast and accurate!

Searching

- **Lecture 1: Binary search applied to sorted array requires $\log_2 N$ operations**

- **Can we do better?**

Searching

- Visitors are identified by IP addresses
  - IP addresses are 4 8-bit numbers which (in base-10) take the form: 251.31.241.80 and identify a user's location on the internet
  - There are 256^4 possible addresses so it's not feasible to maintain an array for all possible addresses
  - Could maintain a sorted list but inserting new addresses would require O(N) operations (in Python)

Searching

- **Visitors are identified by IP addresses**
  - **IP addresses are 4 8-bit numbers which (in base-10) take the form: 251.31.241.80 and identify a user's location on the internet**
  - **There are 256^4 possible addresses so it's not feasible to maintain an array for all possible addresses**
  - **Could maintain a sorted list but inserting new addresses would require O(N) operations (in Python)**

- **Alternate approach: IP addresses are assigned indices based on their values**
  - **A *hash function* takes an address as input and provides and index as output**
  - **The design of hash functions is an important and active subject**
  - **What are the "desirable" properties of a hash function?**

Hash function

- **Consider a hash function** $i = H(a)$
  - i **is a non-negative integer**
  - a **is a numeric representation of an IP address**
  - **Let's say we expect about** 1e3 **unique visitors each month**

- **Then,** H **should distribute these visitors to about** 1e3 **indices**
  - **Desirable property 1: It should always assign a particular IP to the same index on repeat visits**

Hash function

- **Consider a hash function** $i = H(a)$
  - i **is a non-negative integer**
  - a **is a numeric representation of an IP address**
  - **Let's say we expect about** 1e3 **unique visitors each month**

- **Then,** H **should distribute these visitors to about** 1e3 **indices**
  - **Desirable property 1: It should always assign a particular IP to the same index on repeat visits**

- **Since there are** 256^4 **possible IPs, it is impossible to design H so that unique visitors are always assigned unique indices**
  - **When two or more different IPs are assigned the same index, we have a *hash collision***
  - **Say** $0<=i<=N$**. Desirable property 2: *Probability of 2 distinct visitors being assigned same index,*** i ***should be*** $1/N$

Hash function

An example:

- **The IP address is four integers,** a_1,a_2,a_3,a_4
- **Choose four arbitrary integer weights,** w_1,w_2,w_3,w_4

- **Is:** $i = \Sigma w_j a_j$ **a suitable hash function?**

Hash function

An example:

- **The IP address is four integers,** a_1,a_2,a_3,a_4
- **Randomly choose four arbitrary integer weights,** w_1,w_2,w_3,w_4

- **Is:** $i = \Sigma w_j a_j$ **a suitable hash function?**

- **Not quite:**
  - **This can generate** 256^4 **indices, we only want about** 1e3

- **What about** $i = (\Sigma w_j a_j) \bmod 1000$**?**
  - **We'll now have the right number of indices**
  - **But if there are patterns in the IPs, could have large number of visitors assigned the same index**

- **Better to use a prime number,** $i = \Sigma w_j a_j \bmod 997$

Hash function

Better to use a prime number, $i = h(a_j) = \Sigma w_j a_j \bmod 997$

- **How well does this function work?**
- **Given two different addresses** a **and** b**, what is the probability that** $h(a) = h(b)$**?**

Let's assume that $a_4 \neq b_4$ **and that the weights are 4 random integers** mod 997

- **There will be a hash collision if:** $\sum_{j=1}^{4} w_j a_j \equiv \sum_{j=1}^{4} w_j b_j \pmod{997}$

where $x \equiv y \pmod{\mathrm{n}}$ if n divides $x - y$

- **Rearranging:** $\sum_{j=1}^{3} w_j (a_j - b_j) \equiv w_4 (b_4 - a_4) \pmod{997}$
- **The LHS is just some, integer,** c**, and we need to "solve" the following congruence for** w_4: $w_4(b_4 - a_4) \equiv c \pmod{997}$

Hash function

$$w_4(b_4 - a_4) \equiv c \pmod{997}$$

To solve this for w_4, need a result from modular arithmetic:

Let p *be a prime. If* k *is not a multiple of* p*, then there exists an integer* $k^{-1} \in \{1, 2, \ldots, p-1\}$ *such that:* $k \cdot k^{-1} \equiv 1 \pmod{p}$

Hash function

$$w_4(b_4 - a_4) \equiv c \pmod{997}$$

To solve this for w_4, need a result from modular arithmetic:

Let p *be a prime. If* k *is not a multiple of* p*, then there exists an integer* $k^{-1} \in \{1, 2, \ldots, p-1\}$ *such that:* $k \cdot k^{-1} \equiv 1 \pmod{p}$

In our example, $p=997$, $k = (b_4-a_4)$
so, $w_4 \equiv c(b_4 - a_4)^{-1} \pmod{997}$

We *choose:* $w_4 = c(b_4-a_4)^{-1} \bmod 997$
and more generally, require: $w_i \in \{0, 1, ..., p-1\}$

This allow us to state that there is only one w_4 in the range [1 p-1] for which a and b will be assigned the same index.
So, probability(h(a)=h(b)) = 1/p.

Hash table

- **We now have a hash function that will assign indices to IP addresses**

- **What is the general workflow?**

  0. Initialize a dictionary or list (a *hash table)* where you will store addresses
  1. **Given an IP, compute an index**
  2. **And store the IP in the corresponding location in the hash table**
  3. **Append the IP if there is already an address in the computed list**

Hash table

- We now have a hash function that will assign indices to IP addresses

- What is the general workflow?

  0. Initialize a dictionary or list (a *hash table)* where you will store addresses
  1. Given an IP, compute an index
  2. And store the IP in the corresponding location in the hash table
  3. Append the IP if there is already an address in the computed list

- Let's assume that we have M IP addresses and N indices with M>N

- On average, each index will be assigned M/N IPs

- Should choose M to be a prime number close to N (in practice, maybe close to 2N)

Hash table

We started looking for something faster than binary search

What is the cost of using a hash table?

- ***Search* (lookup): Given an IP, find it in the table**
  - **Evaluate the hash function and obtain an index**
  - **Check the number of items stored at index, i.**
  - **If $i > 1$, iterate through items**
  - **Overall, $O(\max(1,M/N))$**
  - **For well-designed hash table and function, will be close to $O(1)$!**

- ***Insert*: Add an IP to table**
  - **Search**
  - **Then append at computed index if IP is not already present**
  - **Overall, close to $O(1)$ – for a careful implementation**

- ***Delete*: Remove an IP**
  - **Same as Insert, just replace "append" with "delete"**

Hash table

Summary:

- **Binary search:** $O(\log_2 N)$ **but requires the maintenance of a sorted list/array**

- **Hash table:** $O(1)$ **for search as well as maintenance**
  - **Provided that the hash function and table are both well-designed!**

Hash table

Summary:

- **Binary search: $O(\log_2 N)$ but requires the maintenance of a sorted list/array**

- **Hash table: O(1) for search as well as maintenance**
  - **Provided that the hash function and table are both well-designed!**

- **What does this look like in Python?**

  - **Let’s move away from the IP problem to a more general task**

  - **Consider input that may be real numbers or even non-numeric**

Python hash function

Python provides a hash function that returns an integer for (almost) any input

- **For an integer,** i, hash(i) = i
- **For two inputs if** a==b **then** hash(a)=hash(b)
  - hash(3.0) = hash(3) = 3
- **For non-integers, the function is less predictable:**

```
In [1]: hash('m3sc')
Out[1]: 5237192937700339269

In [2]: hash('m3c')
Out[2]: 1091467703030128764
```

Python hash table

Can use list of lists to build hash table

- **Compute hash for each item of interest**
- **Store item using hash value as index for list**
- **For hash collisions, append item at index**
- **Lookup:**
  - **Compute hash value**
  - **Search through sub-list at index for item**
- **Not too difficult – try it!**

Python dictionaries

- **But we don't have to build our own hash table!**

- **Python dictionaries *are* hash tables**
  (technically, they are "associative arrays")

- **Dictionaries are containers where each element is a key-value pair**
  - **A key can be considered a label or id**
  - **Example:**

```
In [6]: key = "123.45.241.12"

In [7]: value=[14,1,2019,20]

In [8]: d = {key:value}

In [9]: d
Out[9]: {'123.45.241.12': [14, 1, 2019, 20]}

In [10]: d["123.45.241.12"]
Out[10]: [14, 1, 2019, 20]
```

Python dictionaries

- Dictionaries are containers where each element is a key-value pair
  - A key can be considered a label or id
  - Example:

```
In [6]: key = "123.45.241.12"

In [7]: value=[14,1,2019,20]

In [8]: d = {key:value} #or d[key]=value

In [9]: d
Out[9]: {'123.45.241.12': [14, 1, 2019, 20]}

In [10]: d["123.45.241.12"]
Out[10]: [14, 1, 2019, 20]
```

  - Python applies a hash function to keys to know where to store them and where to look for them

Python dictionaries

- **Important dictionary operations**

***Constant time* O(1):**

```
In [36]: d = dict()     #initialize a new dictionary

In [37]: d[key] = value #associate key with value and store in d

In [38]: d[key]  #value associated with key in d, raises KeyError if key has not been added to d
Out[38]: [14, 1, 2019, 20]

In [39]: x=1

In [40]: d.get(key,x) #value associated with key if key is present, otherwise x
```

Python dictionaries

- **Important dictionary operations**

***Constant time* O(1):**

```
In [36]: d = dict()     #initialize a new dictionary

In [37]: d[key] = value #associate key with value and store in d

In [38]: d[key]  #value associated with key in d, raises KeyError if key
has not been added to d
Out[38]: [14, 1, 2019, 20]

In [39]: x=1

In [40]: d.get(key,x) #value associated with key if key is present,
otherwise x

In [42]: key in d   #is key in d?
Out[42]: True

In [43]: len(d)   #number of key-value pairs in d
Out[43]: 1

In [44]: del d[key]  #remove key (and its associated value) from d
```

Python dictionaries

- **Important dictionary operations**

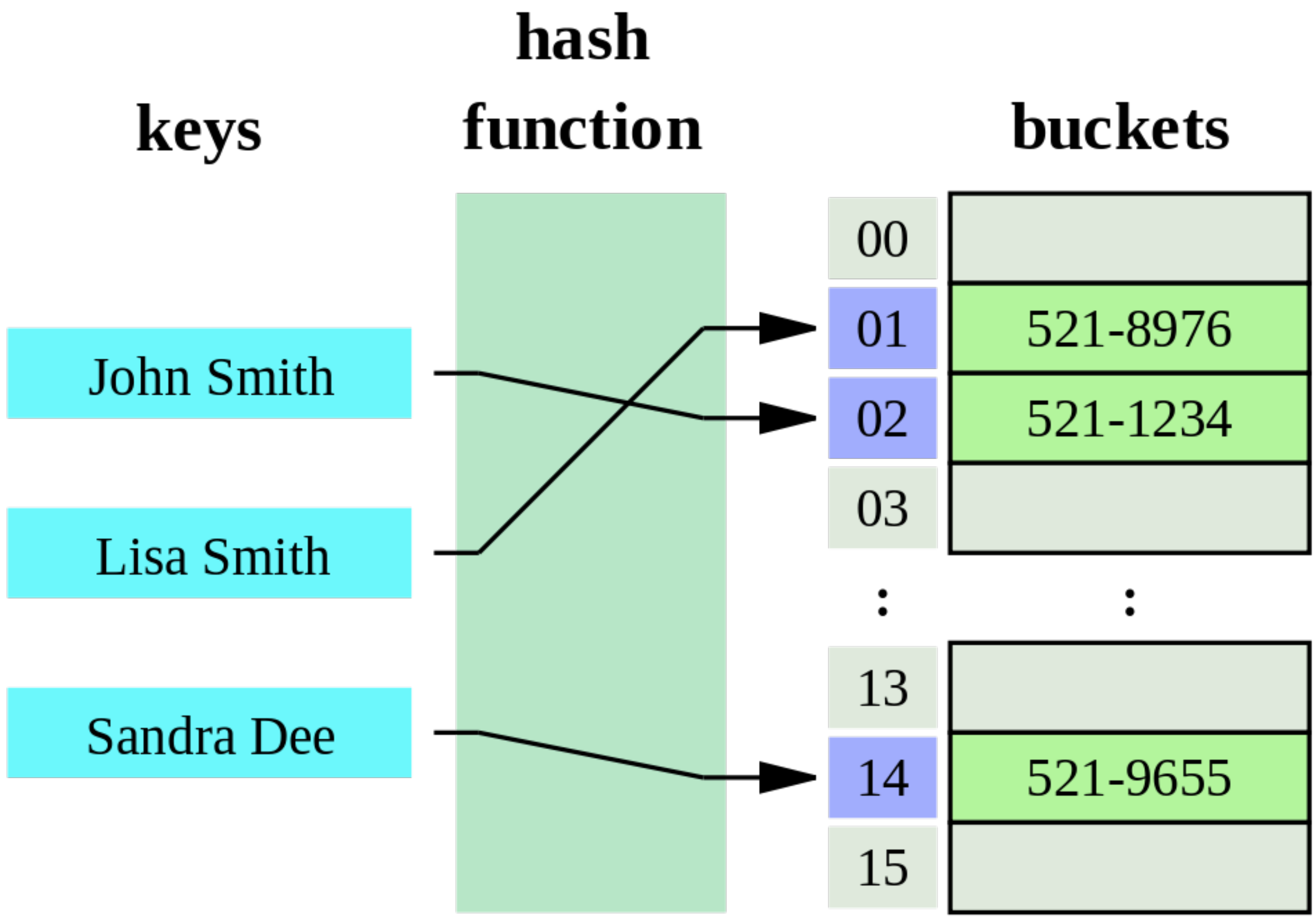
***Constant time* O(1):**

```
In [36]: d = dict()     #initialize a new dictionary

In [37]: d[key] = value #associate key with value and store in d

In [38]: d[key]  #value associated with key in d, raises KeyError if key has not been added to d

In [40]: d.get(key,x) #value associated with key if key is present, otherwise x

In [42]: key in d   #is key in d?

In [43]: len(d)   #number of key-value pairs in d

In [44]: del d[key]  #remove key (and its associated value) from d
```

***linear time* O(N):** `In [45]: for key in d:   # iterate over keys in d`

This is exactly what we need to build and maintain a hash table.

Python hash table



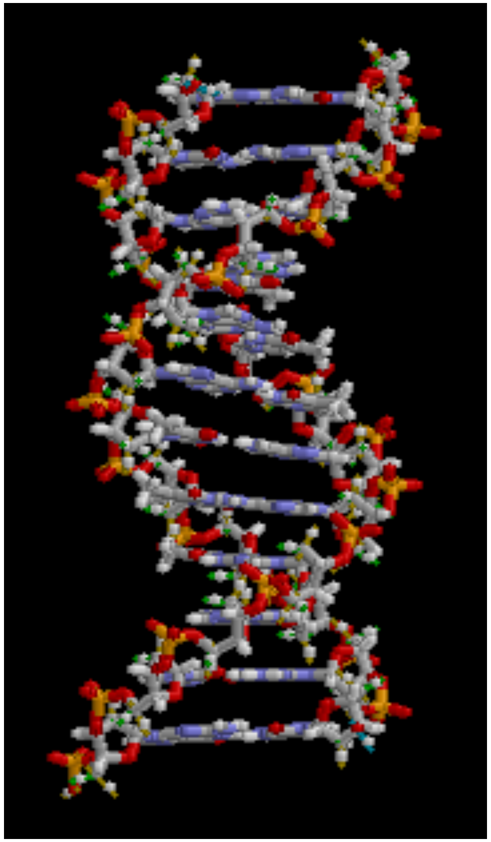
Genetic code

- DNA (and RNA) consists of two “strands” arranged in a double helix and connected with covalent bonds

- Each strand contains a sequence of nucleotides

- DNA is constructed from 4 nucleotides:
  - Adenine
  - Cytosine
  - Guanine
  - Thymine

  (RNA has Uracil in place of Thymine)



from wikipedia

Genetic code

- DNA is constructed from 4 nucleotides (or *bases*):
  - Adenine
  - Cytosine
  - Guanine
  - Thymine

  (RNA has Uracil in place of Thymine)

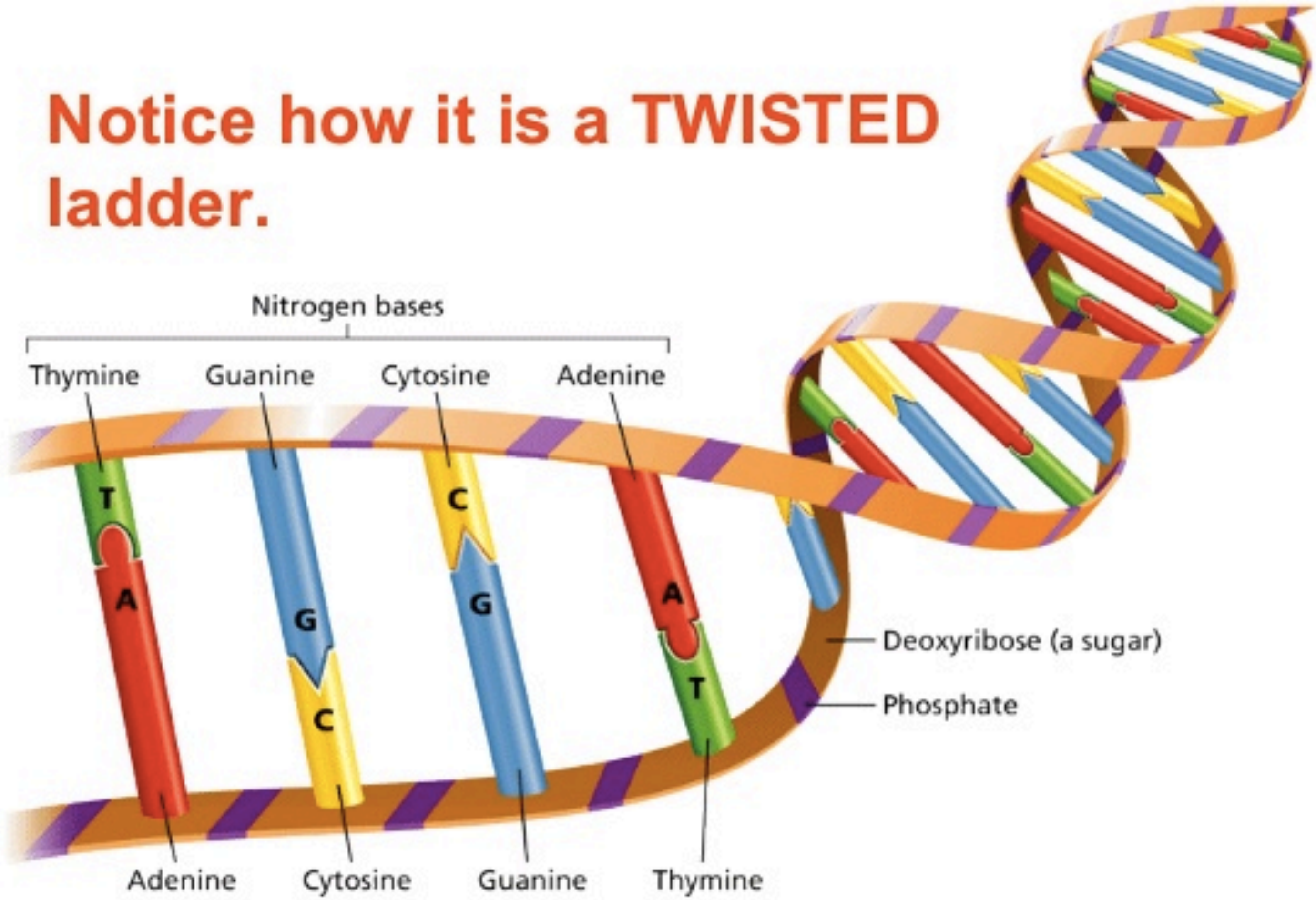
- Adenine bonds with Thymine and Guanine bonds with Cytosine

- So if one strand contains the sequence GCTTCA the other strand will contain CGAAGT in the corresponding location

- During cell division, each daughter cell gets one strand
  - And then the needed 2nd strand can be constructed so A's pair with T's and C's pair with G's

Here is a DNA Molecule

Notice how it is a TWISTED ladder.



Genetic code

- DNA is constructed from 4 nucleotides (or *bases*):
  - Adenine
  - Cytosine
  - Guanine
  - Thymine

  (RNA has Uracil in place of Thymine)

- *Codons* consist of three DNA bases and contain code for synthesizing amino acids
  - Proteins are built from amino acids
  - 64 possible codons, but there are only 20 essential amino acids specified by DNA

- Gene sequencing involves:
  - Extracting the sequence of bases from DNA samples
  - Investigating the proteins or functions associated with codons and their sequences

Pattern finding

- A fundamental computational problem:
  - Given a DNA sequence, search for a pattern
  - Both the sequence and the pattern can be extremely long
  - The number of patterns can also be large
- Fruit fly genome: 139.5 million base pairs

- Humans: 3 billion or 6 billion pairs (depending on the cell type)

- How can we *efficiently* search for patterns?

Pattern search

- **Problem setup:**
  Specify a N-character sequence, *S*, and a M-character pattern, *P*
  Find all locations in *S* where *P* occurs

- **Example:**

  ***S* = ATGTTGTACCGTATCGG**
  ***P* = GTA**

  N=16, M=3