Algorithms for Data Science

Daniel Lawson — University of Bristol

Lecture 09.2 (v2.0.0)

Psst! Want some Big Data?

Questions

- Can we quickly tell if we've seen data before?
- How quickly can we access it?
- How can we randomly sample from a near-infinite data stream?
- Can we count things without storing them all?

Hash functions

- One of the most important components in good algorithmic design is the **hash**.
- Simply, a hash h is a map for $h(x) = u$ with:

$$x \in \mathcal{X} \to u \in \mathcal{U}[0, r).$$

- i.e., we map each item in the space $\mathcal{X}$ into the Uniform distribution on the integers $0, \ldots, r-1$.
- Each item will always map to the same integer.

Hash examples

- Some simple methods for creating keys from integers.
- Open DSA - Data Structures and Algorithms is a great reference.
- Modulo r

```
x % 16 # modulo 16
```

Hash examples

- Some simple methods for creating keys from integers.
- Open DSA - Data Structures and Algorithms is a great reference.
- Modulo r

```
x % 16 # modulo 16
```

- Binning (floor function or integer division)

```
x // 32 # need to know max(N) for r
```

Hash examples

- Some simple methods for creating keys from integers.
- Open DSA - Data Structures and Algorithms is a great reference.
- Modulo r

```
x % 16 # modulo 16
```

- Binning (floor function or integer division)

```
x // 32 # need to know max(N) for r
```

- Mid-Square method: square the value, use the middle digits in the hash

Hash considerations

- There are many choices for a hash function in practice. Considerations include:
- **Randomness**. For many applications (e.g. cryptography) we want no correlation between x and u.
- **Locality**. For other applications (e.g. locality sensitive hashing) we want similar x to produce similar u.
- **Collisions**. We may wish to reduce collisions on a subset of the potential input space. For example, if $x \in [0, r)$ and $u \in [0, r)$ it is possible to eliminate collisions.
- **Compute**. Hash functions vary in their compute cost.
- **Families**. It is often useful to be able to index a family of hash functions with the same computational cost that return different values.

Data Structures

- Data structures are representations of a **set** of data
- This representation is particularly important when sets are **dynamic**, i.e. grow or shrink
- We will perform **operations** on the set, which will have an associated computation cost
- The data structure has an associated space cost
- Making the right choice of data structure is an essential component of data science

Fixed size elementary data structures

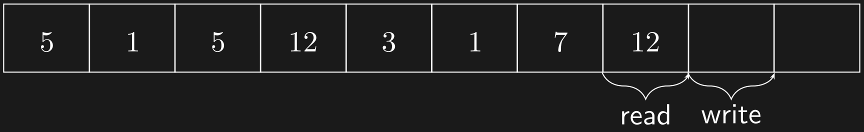
- We are familiar with the concepts of:
  - **Arrays**: A segment of memory containing n data of the same type
  - **Vectors**: Arrays with additional operations defined
  - **Multi-dimensional arrays**: Arrays of length $n = n_0 \times n_1 \times \cdots \times n_k$, with entries specified according to a protocol (e.g. row-wise)
  - **Matrices/Tensors**: Multidimensional arrays with additional operations defined
- It is clear that arrays are a fundamental concept!

Elementary data structures: Stacks and Queues

5	1	5	12	3	1	7	12		

- **Stacks**: Data are stored in an array using “first in, last out”: insertions and deletions occur at the same end
  - Implemented as a pointer to the last read location
- **Queues**: Data are stored in an array using “first in, first out”: insertions occur one end, deletions the other
  - Implemented as a pointer to the end (for writing) and start (for reading) that tracks removed items

Elementary data structures: Stacks and Queues



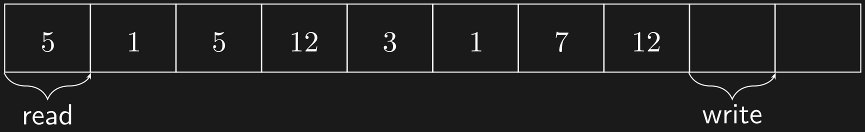
- **Stacks**: Data are stored in an array using "first in, last out": insertions and deletions occur at the same end
  - Implemented as a pointer to the last read location
- **Queues**: Data are stored in an array using "first in, first out": insertions occur one end, deletions the other
  - Implemented as a pointer to the end (for writing) and start (for reading) that tracks removed items

Elementary data structures: Stacks and Queues

5	1	5	12	3	1	7	12		

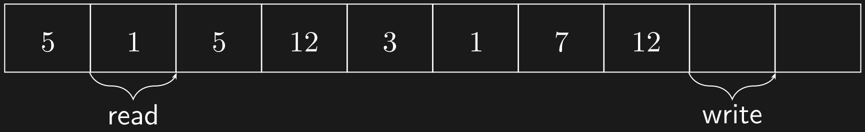
- **Stacks**: Data are stored in an array using “first in, last out”: insertions and deletions occur at the same end
  - Implemented as a pointer to the last read location
- **Queues**: Data are stored in an array using “first in, first out”: insertions occur one end, deletions the other
  - Implemented as a pointer to the end (for writing) and start (for reading) that tracks removed items

Elementary data structures: Stacks and Queues



- **Stacks**: Data are stored in an array using “first in, last out”: insertions and deletions occur at the same end
  - Implemented as a pointer to the last read location
- **Queues**: Data are stored in an array using “first in, first out”: insertions occur one end, deletions the other
  - Implemented as a pointer to the end (for writing) and start (for reading) that tracks removed items

Elementary data structures: Stacks and Queues



- **Stacks**: Data are stored in an array using “first in, last out”: insertions and deletions occur at the same end
  - Implemented as a pointer to the last read location
- **Queues**: Data are stored in an array using “first in, first out”: insertions occur one end, deletions the other
  - Implemented as a pointer to the end (for writing) and start (for reading) that tracks removed items

Elementary data structures: Stacks and Queues

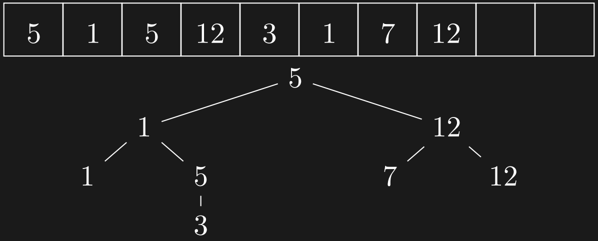
5	1	5	12	3	1	7	12		

- **Stacks**: Data are stored in an array using "first in, last out": insertions and deletions occur at the same end
  - Implemented as a pointer to the last read location
- **Queues**: Data are stored in an array using "first in, first out": insertions occur one end, deletions the other
  - Implemented as a pointer to the end (for writing) and start (for reading) that tracks removed items
- Despite implementation similarities, both have different Data Science properties!

Elementary data structures: Linked List

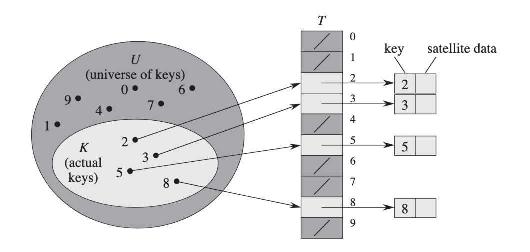
- **Linked list**: Data are stored in a list, with a pointer to the location of the next item
    - Fast traversion, insertion and deletion
    - Slow random access
    - Can be doubly linked

Elementary data structures: Binary Trees & Heaps



- **Binary Trees**: Data are stored in a **binary** linked list, i.e. each node has (up to) two children
  - Data can be stored at nodes or leaves
  - **Critical** to define the left/right operation!
- Position is decided by a key, which can be related to the value
  - In the picture, values $\leq x$ go left, $> x$ go right
  - Some binary tree structures assign values to internal nodes, e.g. means/ranges
- **Heaps**: A binary tree where each node's key is (larger) than it's children

Elementary data structures: Hash Tables



- **Hash Tables**: Data location determined by the **key**
- The key is a **hash** $x = h_l$: either of an attribute (e.g. a name), or of the value
- Advantage is $O(1)$ lookup cost. Usage is:
  1. Compute $u = h_2(x)$
  2. Set $u' = u\%r$
  3. To insert: store y at this position. On collision, we use some rule to find an empty space, such as rehashing, or storing a linked list.
  4. To lookup: retrive this value (using the same rule about collisions).

Sampling (for big data)

- If there are N (large) items, how do we correctly sample n of them?
- Naive approach: read in the data, choose n at random, done.
- What if the data don't fit in memory? We might choose a subset e.g. by:
  - **Random sampling**: Choose each point with probability $p = n/N$
  - **Uniform sampling**: Choose every n/Nth point
  - Efficiently?

Sampling (when we don't know N)

- **Reservoir sampling**:
  - Keep the first n items. For the remaning items i:
  - Accept the new item with probability n/i
    - discard uniformly from the n.
  - Otherwise, keep the old items.
- Weighted versions etc exist.
- Generates samples uniformly from the whole set of n with fixed storage.

Non-Uniform sampling

- Sometimes, most data is “boring”. We want to sample the “most useful” data.
- One solution is to divide the data into histogram bins and sample inversely with frequency using e.g. reservoir sampling within each
- How to choose the bins?
  - Choice in advance requires knowledge of the data, or looking at it already
  - Dynamic approaches are possible where the bins are learned in a **streaming** manner[1]
  - The algorithm can be tuned for estimating particular quantities, e.g. the mean[2]

[1] Streaming histogram implementation

[2] Risto Tuomainen Data Sampling for Big Data

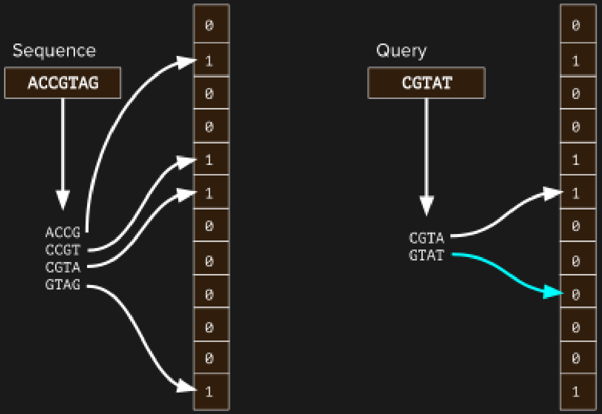
Filtering

- Filters have the goal of retaining information regarding which data have previously been seen, **without storing it** all.
- Example: we have a datastream of (many) observed MAC addresses from users.
  - Question: have we seen value x before?
  - Can we do this with **constant cost** $\Theta(1)$ per item?

Bloom Filter

- A **bloom filter** can tell in constant time whether:
  1. a data point is not in the database
  2. a data point might be in the database
- It does this by storing all of the observed data solely as a hash $h(x) \to (0, r]$.
  - The data are stored as a bitvector $\mathbf{b}_r$.
  - The larger the range, the more precise the answer will be but the greater the cost.
  - For each datapoint x_i we:
  1. Compute k hashes in $[0, r)$, $h_k(x_i)$
  2. Set all bits hashed into to one, i.e. $b_r(h_k(x_i)) = 1$
  - At lookup time: if any $b_r(h_k(x_i)) = 0$ then we have not seen this item before.
- See Bill Mill's excellent Bloom filter practical

Bloom Filter Example



Choosing parameters for a bloom filter

- There are three variables: the **number of data expected** to be stored, n, the **number of hashes** k and the **length of the bitvector** r.
- The **error rate** is expected to be $(1 - \exp(-kn/r))^k$
- It turns out that this is minimised when $k = r/n \ln(2)$
- You then trade of error rate for storage size (for the bit vector) and compute cost (for the hashes)
- Bloom Filters are very useful, for example in Network analysis[3]

[3]Broder & Mitzenmacher "Network Applications of Bloom Filters: A Survey" (2003) Internet Mathematics 1:485-509

Sketching

- Sketching is obtaining the frequency properties of your data from a data stream.
- One important class is probabilistic counting, which addresses how many of each class there are.

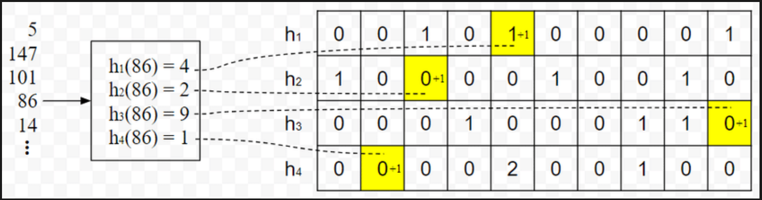
Count-min-sketch

- Count-min-sketch works just like a bloom filter, except that we store an integer for each has rather than a single bit.
- We initialise $\mathbf{b}_r = \mathbf{0}$, and then:
  1. Compute k hashes in $(0, r]$, $h_k(x_i)$
  2. Add one to all bits hashed into, i.e. $b_r(h_k(x_i)) += 1$
- At lookup time, the number of items is estimated to be

$$\mathrm{argmin}_{h_k(x_i)} b_r(h_k(x_i))$$

  i.e. the minimum count.
- See e.g. Python implementation of Count Min Sketch by Rafael Carrascosa (part of PyPI)

Sketching Example



Other important algorithms:

- The **MinHash** algorithm quickly computes similarities between sparse feature vectors such as **documents**.
- **Locality Sensitive Hashing** reduces the dimensionality of data by representing an object as a set of hashes, chosen so that "similar" items have "similar" hash values
- The **Hashing Trick** is a Machine-Learning tool for turning arbitrary objects into features - just take one or more locality sensitive hashes of the object as new features.
- There are a range of sketches with different biases, such as the Count-Mean-Sketch and others[4].

[4] Goyal, Daume & Cormode "Sketch Algorithms for Estimating Point Queries in NLP" (2012) Proc. EMNLP.

MinHash motivation

- Consider a very large, potentially sparse, **binary** feature space for which we have observations $A = \{x_i\}$ and $B = \{x_k\}$. How similar are they?
- One natural measure is the **Jaccard Similarity**:

$$J(x_i, x_j) = \frac{x_i \cap x_j}{x_i \cup x_j}$$

- This is slow to compute with a large sparse features space, such as **words**.
- The solution is to approximate the similarity via MinHash.

MinHash algorithm

- To compute a single MinHash Signature:
  - Use a **random hash function** and apply it to all values in A and B.
  - Compute the minimum of each of these.
  - The probability of these being equal turns out to $J(A, B)$.
- To estimate J, we simply do this several times.
- This was used for website Duplicate detection by AltaVista and was confirmed to be still in use by Google in 2007. There are a lot of websites...
- See e.g. Chris McCormick's Minhash tutorial or the Mining of Massive Datasets book and course.

Discussion

- Exploiting convenient algorithms forms a key part of many high-throughput models.
- You need to do this with big data, to get a smaller dataset you can work with:
  - Many data streams have a **power-law** distribution of activity: much of the data are seen only once, whilst some **heavy hitters** might make up the majority of the dataset.
  - Identification of heavy hitters and singletons allows them to be treated specially which can massively reduce computational burden.
- Remember not to use complicated approximate algorithms if you can simply store everything in memory and count it.

References

- Advanced algorithms:
  - The Mining of Massive Datasets book and course.
  - Risto Tuomainen Data Sampling for Big Data, covering sampling, filtering, sketching, etc.
  - Streaming histogram implementation
  - Bill Mill's excellent Bloomfilter practical
  - Chris McCormick's Minhash tutorial
  - Python implementation of Count Min Sketch by Rafael Carrascosa (part of PyPI)
  - Leo Martel notes on Streaming Data Algorithms which is notes on the paper
  - Cormode's notes on Count-Min Sketch
  - Chakrabarti's Lecture Notes on Data Stream Algorithms
  - Broder & Mitzenmacher "Network Applications of Bloom Filters: A Survey" (2003) Internet Mathematics 1:485-509
  - Geravand & Ahmadi "Bloom filter applications in network security: A state-of-the-art survey" (2013) Computer Networks 57:4047-4064
  - Goyal, Daume & Cormode "Sketch Algorithms for Estimating Point Queries in NLP" (2012) Proc. EMNLP.

References

- Data structures:
  - Cormen et al 2010 Introduction to Algorithms is very accessible and recommended for data structures.
  - Open DSA - Data Structures and Algorithms.