Computation in the Data Stream model

Making the most of limited resources

Julián Mestre

School of Information Technologies
The University of Sydney

THE UNIVERSITY OF SYDNEY

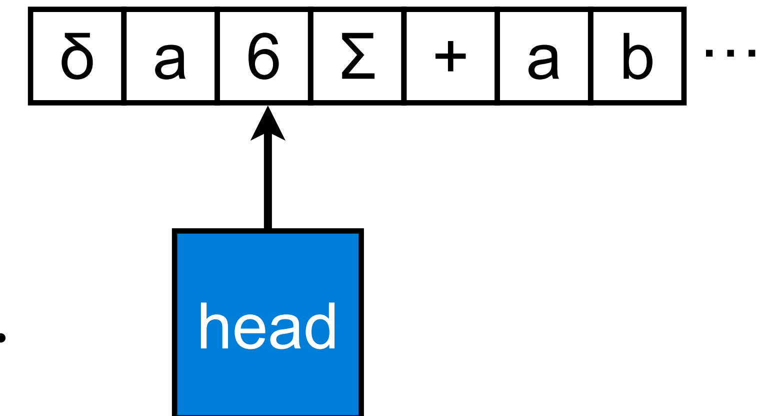
Computational models

Theoretical computer science deals with the study of what a computer *can* and *cannot* do.

Rather than working with real computers we work with abstract computer models that allow us to make useful predictions about the real world.

Now and then our models need refining.

Turing machines



Ingredients:

- Infinite tape with symbols written on it.
- Head that can read/write symbols and move left/right.
- State register.
- Action table that tell the machine what to do.
- Measure of efficiency: number of steps.

Introduced by Alan Turing to explore the limits of computability/logic.

Not an accurate model of modern computers

THE UNIVERSITY OF SYDNEY

Random Access Memory model

Ingredients

- Programs that use a limited set of operations, e.g., { +, *, -, =, if }.
- Memory (as much as we need). Each position stores a number.
- Measure of efficiency: # operations and # memory accesses.

Variants of the basic model can deal with:

- Memory hierarchies.
- Parallel processors.
- Bounded precision memory.

Fairly accurate model...

until you have to handle truly massive data sets.

Big data

Internet routers forward IP packets that have origin and destination addresses and carry data.

You may want to compute:

- How many packets went through?
- How many different origin-destination address pairs were there?
- Report addresses responsible for more than 1% percent of total traffic.
- Same queries but within a time window

In general, it is not feasible to store all though traffic and then use standard algorithms from the RAM model.

THE UNIVERSITY OF SYDNEY

Data streaming model

Ingredients:

- Similar to RAM model but with limited memory.
- Instance is made up of *items*, which we get one by one.
- Instance is too big to fit into memory.
- We are allowed several passes over the instance .

Measure of efficiency:

- Time complexity: processing time per item.
- Space complexity: amount of memory used.
- Pass complexity: # of passes over stream.

What are the trade-offs between these different measures?

THE UNIVERSITY OF SYDNEY

Majority problem

Let $a_1, a_2, ..., a_m$ be a sequence of numbers in the range $\{1, 2, .., n\}$ such that $m \geq n$

Def.: The sequence $a_1, a_2, ..., a_m$ has a *majority element* if at least $m/2 + 1$ entries are the same.

Def.: The *majority problem* is to determine if a given sequence has a majority element.

What's an obvious algorithm for this problem?

What's the running time? How much memory does it use?

Algorithm 1

For each number x in $\{1, ..., n\}$ keep $count(x)$ of how many entries in the sequence equal so far.

When we get a new item a_i, increment $count(a_i)$.

Once we get a item x causing $count(x) > m/2$ stop.

Complexity analysis:

- Time complexity: constant time per item.
- Space complexity: "roughly" $n \log m$ bits
- Pass complexity: one.

Algorithm 2

For each x in $\{1,\ldots, n\}$ do a dedicated pass to compute count(x).

Once we find an x with count(x)>$m/2$ stop.

Complexity analysis:

- Time complexity: constant time per item.
- Space complexity: "roughly" $\log m$ bits
- Pass complexity: n.

We traded passes for space. Is there a better trade-off?

THE UNIVERSITY OF SYDNEY

First the bad news

Consider a sequence of length $2n$ ($m=2n$) where

- first n numbers encode a subset of size $< n$,
- last n numbers are all the same.

Examples for $n=5$:

- An instance with no majority element:

1	2	3	5	5	2	2	2	2	2

- An instance with majority element:

1	3	4	4	4	2	2	2	2	2

Not enough memory to distinguish between the two!

Thm. Any single pass algorithm must use at least n bits of memory

And now the good news: Algorithm 3

There is a 2-pass algorithm that uses very little memory and does constant amount of work per item!

Main idea:

- Do one pass to identify a potential majority element x.
- Do an extra pass to confirm that x is indeed a majority element.

Algorithm 3

In the first pass:

- Keep a single *counter* associated with an *item value*
- Upon getting item y we check:
  - If the counter is 0, reset the value to y and increment counter by 1
  - If the counter > 0 and the associated value ≠ y, decrease counter by 1
  - If the count > 0 and the associated value = y, increase counter by 1

In the second pass:

- Let x be the value associated with the counter at the end of the first pass
- Scan stream to compute count(x) and check whether count(x) > m/2

THE UNIVERSITY OF SYDNEY

Analysis: Correctness

Obs.: If there is a majority element then Algorithm 3 will identify it during the first pass.

Imagine augmenting the algorithm so that when we decrease the counter associated with x because we got $y \neq x$, output pair (x,y).

Suppose x is a majority element but Algorithm 3 fails to identify it.

Then it must produce $> m/2$ pairs of the form (x, y) for $y \neq x$.

But that's means there are more than m elements in the stream. A Contradiction! So our observation follows.

THE UNIVERSITY OF SYDNEY

Analysis: Complexity

Obs.: Algorithm 3 uses "roughly" log m bits of memory

Obs.: Algorithm 3 uses constant amount of time per item.

Obs.: Algorithm 3 does two passes over the instance.

Thm. There is a 2-pass, constant time per item, O(log m) bit streaming algorithm for the majority problem.

THE UNIVERSITY OF SYDNEY

Data streaming model

A fairly recent model of computation that has led to many exciting developments

Usually we allow streaming algorithms to return approximations

To deal with bad instances we use randomization

THE UNIVERSITY OF SYDNEY

Bonus topic: Sampling

Sampling is a basic primitive that is used in many algorithm.

Given a sequence $a_1, a_2, ..., a_n$ of distinct numbers, we would like to pick x at random so that $Pr[x = a_i] = 1/n$ for all $i = 1,..., n$.

In the data streaming version we are shown items one at a time and we want to keep a sample with the correct probability.