DCGI

KATEDRA POČÍTAČOVÉ GRAFIKY A INTERAKCE

MODERN ALGORITHMS (not only in computational geometry)

PETR FELKEL

FEL CTU PRAGUE

felkel@fel.cvut.cz

https://cw.felk.cvut.cz/doku.php/courses/a4m39vg/start

Based on [Kolingerova], [Brönnimann], and [Muthukrishnan]

Version from 19.12.2019

Modern algorithms

1. Computational geometry today
2. Space efficient algorithms
   (In-place / in situ algorithms)
3. Data stream algorithms
4. Randomized algorithms

Computational geometry today

- Popular: beauty as discipline, wide applicability
- Started in 2D with linear objects (points, lines,…), now 3D and nD, hyperplanes, curved objects,…
- Shift from purely mathematical approach and asymptotical optimality ignoring singular cases
- to practical algorithms, simpler data structures and robustness => algorithms and data structures provable efficient in realistic situations (application dependent)

Space efficient algorithms

Space efficient algorithms

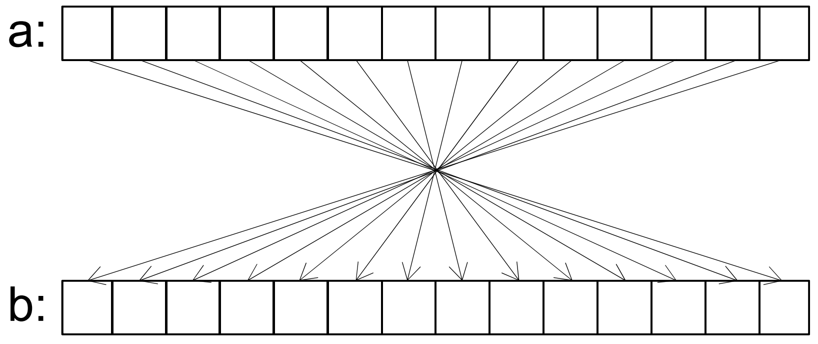
- output is in the same location as the input and
- need only a small amount of additionally memory
  - *in-place* – O(1) extra storage
    sometimes including O(*log n)* bits for indices
  - *in situ* – O(*log n*) extra storage

Space efficient algorithms - practical advantages

- Allow for processing larger data sets
  - Algorithms with separate input and output need space for 2n points to store – O(*n*) extra space
  - Space efficient algs. – O(1) or O(log *n*) extra space
- Greater locality of reference
  - Practical for modern HW with memory hierarchies (e.g., registers – ram on chip (caches) – main RAM, disk latency, network latency)
- Less prone to failure
  - no allocation of large amounts of memory, which can fail at run time
  - good for mission critical applications
- Less memory => faster program

Ex: String reverse

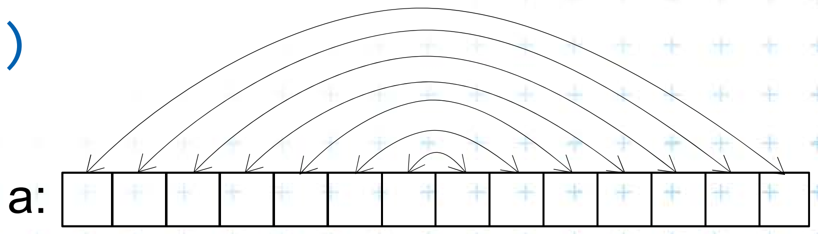
```
function reverse(a[0..n])
    allocate b[0..n]
    for i from 0 to n
        b[n-i] = a[i]
    return b
```



$O(n)$ extra storage

×

```
function reverseInPlace(a[0..n])
    for i from 0 to floor(n/2)
        swap (a[n-i], a[i])
```

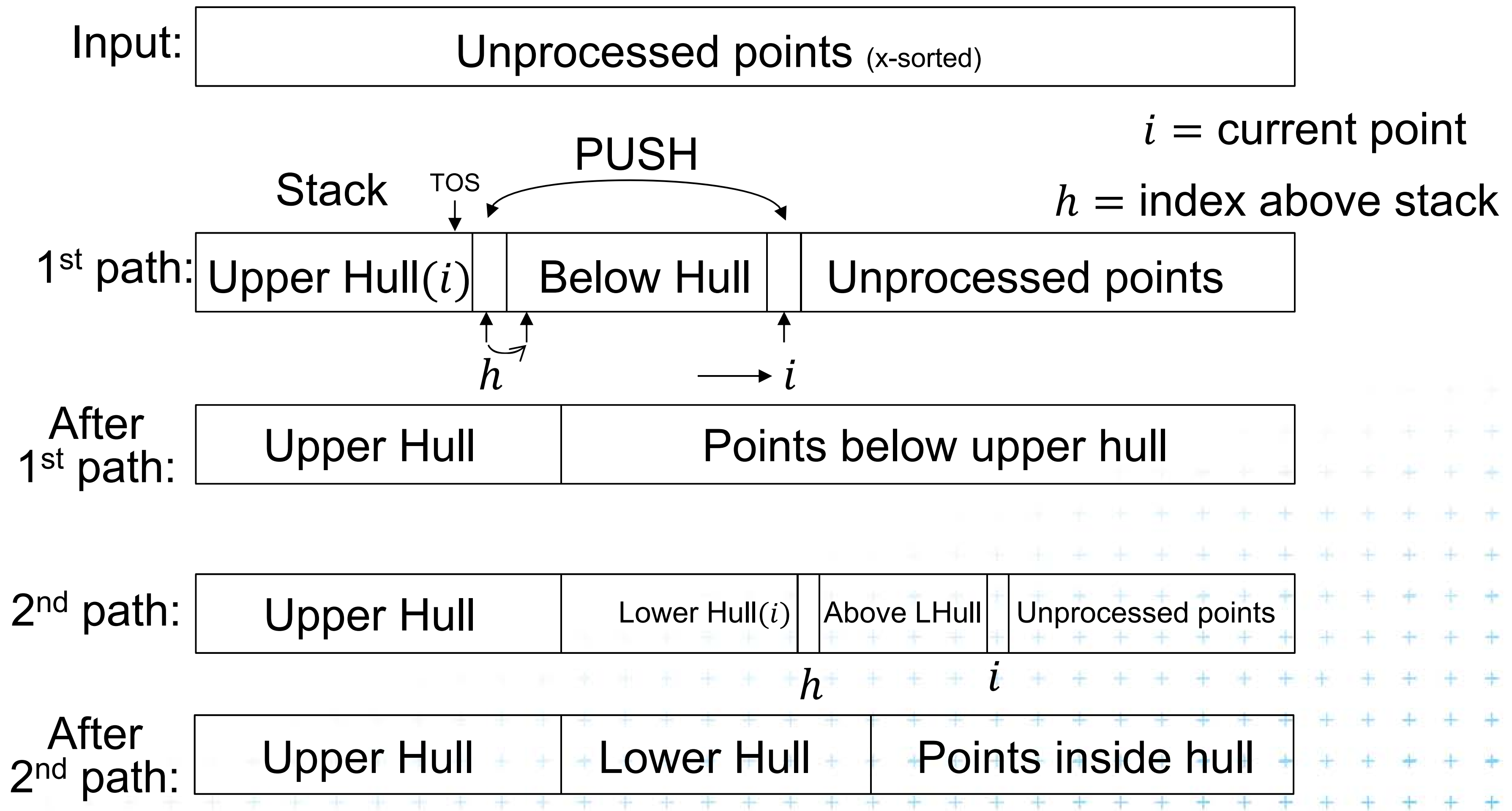


$O(1)$ extra storage

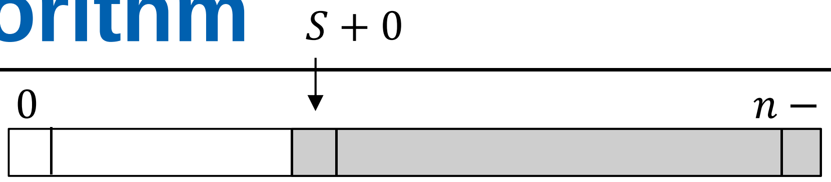
In-place sorting

- In array – continuous block in memory
  - nth element in $O(1)$ time
  - Select sort, insert sort … in-place,
    $O(1)$ additional memory, $O(n^2)$ time
  - Heapsort – in-place, $O(1)$ add. memory, $O(n \log n)$ time
  - Quicksort – in-situ, $O(\log n)$ add. memory for recursion
  - Mergesort – not in-place, not in-situ, $O(n)$ add. memory
- In list – linked lists in dynamical memory
  - nth element in $O(n)$ time
  - Mergesort –in-situ, $O(\log n)$ add. memory, $O(n \log n)$ time

Graham in-place algorithm principle



Graham in-place algorithm



Graham-InPlaceScan(S, n, d)
Input: S – pointer to array of length n with points in plane, $d = \pm 1$ direction
Output: Upper or lower Convex Hull in clockwise order

```
                                             // d controls the sort direction:
1.  InPlace-Sort($S, n, d$)                  //  $d = 1$ sort ascending   for upper hull
2.  $h \leftarrow 1$   // 1st point in stack   //  $d = -1$ sort descending for lower hull
3.  for $i \leftarrow 1 \dots n-1$ do
                                                  TOS-1          TOS        NEW
4.      while $h \geq 2$ and not right turn( $S[h-2]$, $S[h-1]$, $S[i]$ ) do
5.          $h \leftarrow h - 1$          // pop top element from the stack
6.      swap $S[i] \leftrightarrow S[h]$  } // push the new point to the stack
7.      $h \leftarrow h + 1$              } // increment stack length
8.  return $h$                      // end of convex hull (the first point above the stack)
```

The array: S = pointer to the sub-array (to its first point)
h = index of the first point above the stack (offset to S)
i = index of the current point

Graham in-place algorithm

```
Graham-InPlaceHull($S, n$)
Input:    $S$ – an array of length $n$ with points in plane
Output:   Convex Hull in clockwise order (CW)
                                                              sort direction
                                                                O(n log n)
1.   $h \leftarrow$ Graham-InPlaceScan($S, n$, 1)    // 1= ascending – CW upper hull
2.   for $i \leftarrow 0\,..\,h-2$ do
3.        swap $S[i] \leftrightarrow S[i+1]$             // bubble $a$ to the right O(h)
4.   $h' \leftarrow$ Graham-InPlaceScan($S+h-2, n-h+2, -1$)  // lower hull
5.   return $h+h'-2$
```

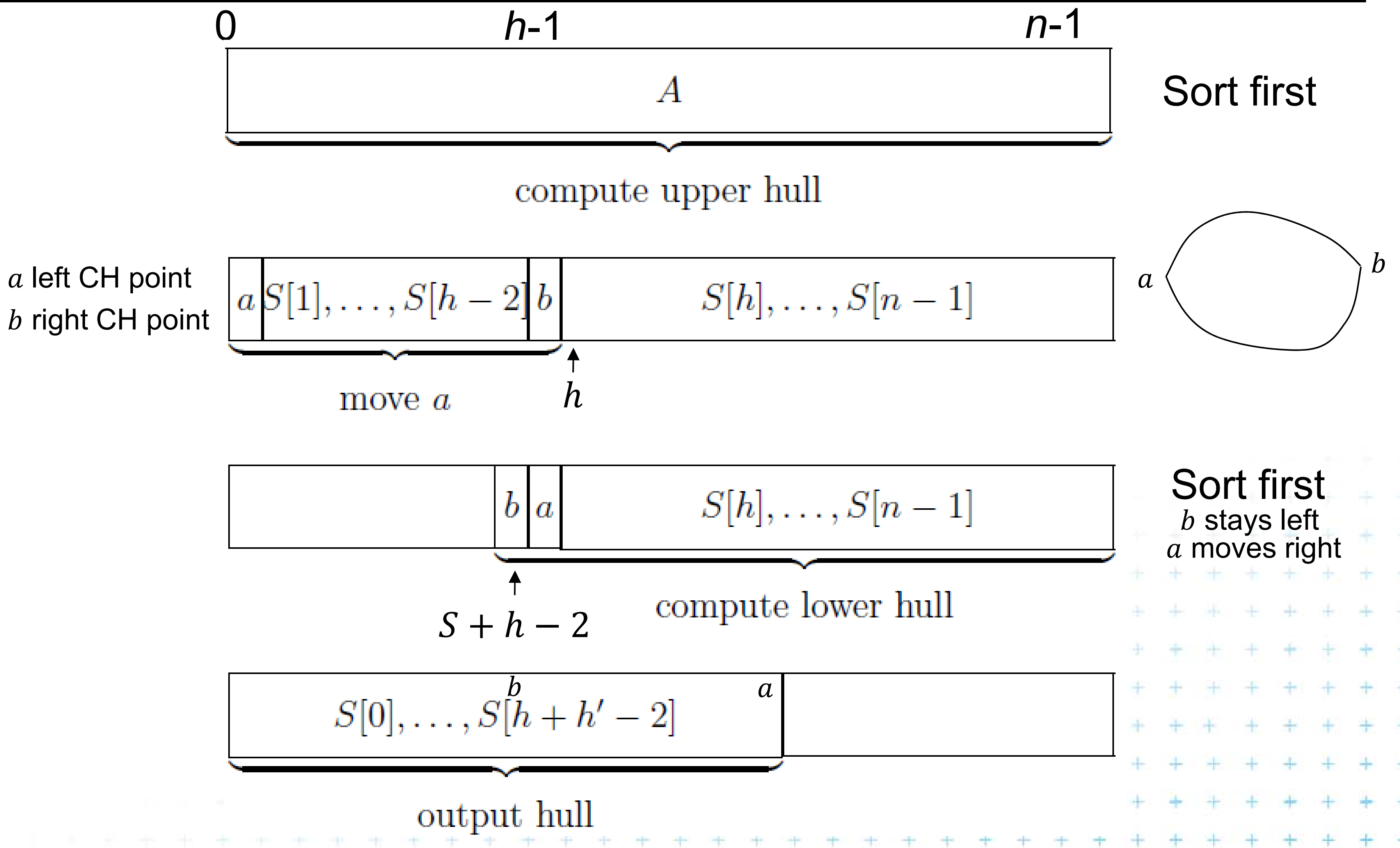
Principle:

Stack at the beginning of the array S on indices $[0\,..\,h-1]$

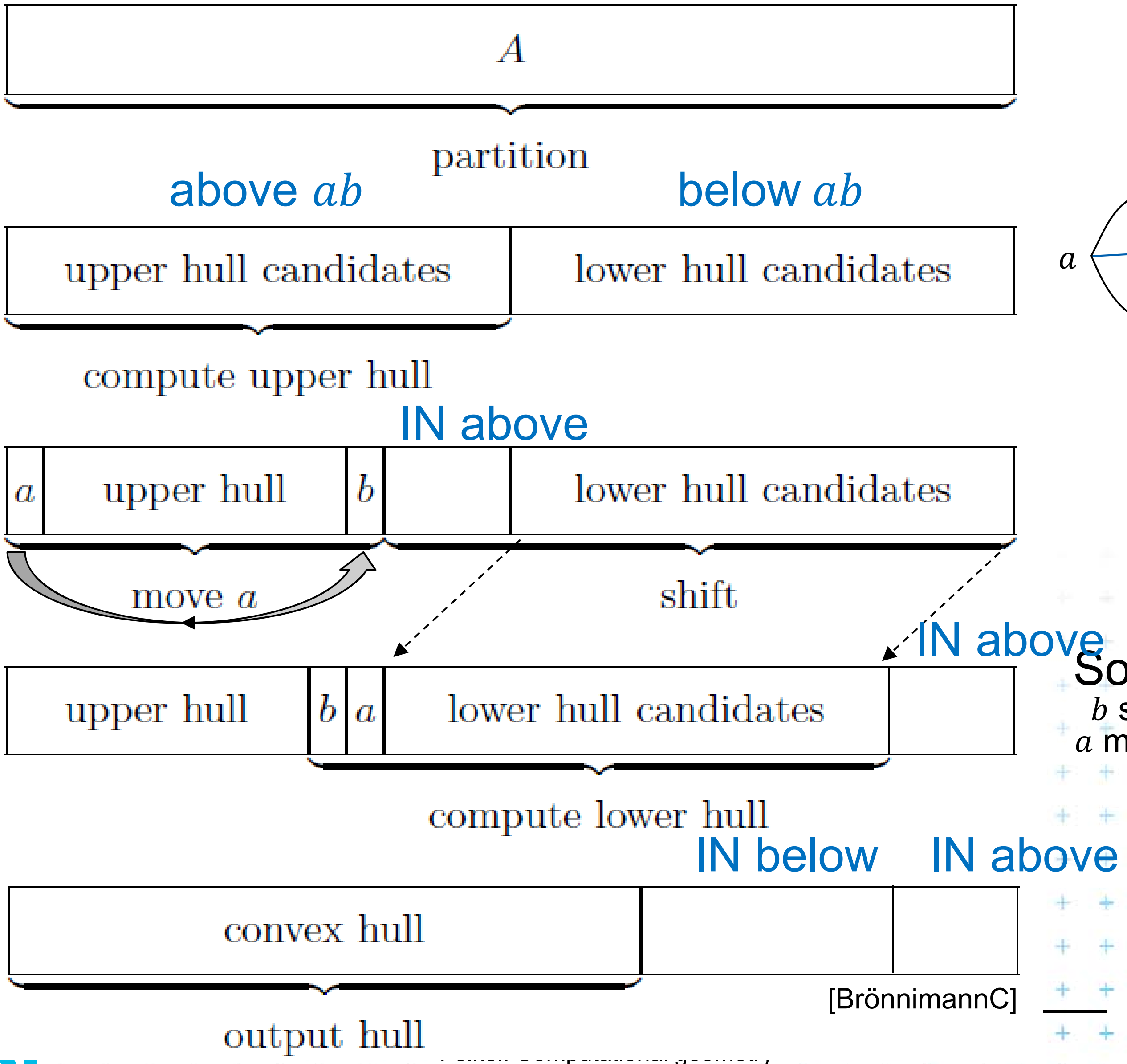
Exchange by swap operation

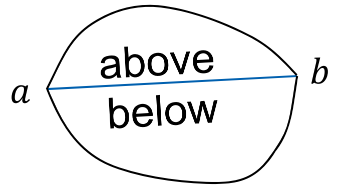
We need the in-place sort

Graham in-place algorithm



Optimized Graham in-place algorithm





Sort first:
b stays left
a moves right

[BrönnimannC]

Data stream algorithms

|data| $\gg$ |RAM or disk|

Data stream algorithms [Indyk]

- Data stream = a massive sequence of data
  - Too large to store (on disk, memory, cache,…)
- Examples
  - Network traffic
  - Database transactions
  - Sensor networks
  - Satellite data feeds
  - …
- Approaches
  - Ignore it (CERN ignores 9/10 of the data)
  - Develop algorithms for dealing with such data

Motivation example [Muthukrishnan]

- Paul presents numbers $x = \{1 \dots n\}$ in random order, one number missing
- Carole must determine the missing number but has only $O(\log n)$ bits of memory

Any idea?

- Compute the sum of the numbers and subtracts the incoming numbers one by one.

$$missing\ number = \frac{n(n+1)}{2} - \sum_{i<n} x[i]$$

- The missing number "remains"

Motivation example [Muthukrishnan]

- And two missing numbers i, j ?

- Store sum of numbers s and sum of squares s'

$$i + j = \frac{n(n+1)}{2} - s$$
$$i^2 + j^2 = \frac{n(n+1)(2n+1)}{6} - s'$$

(this principle is applicable for k-missing numbers)

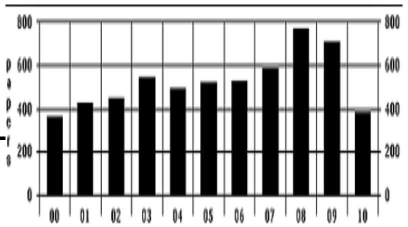
Basic data stream model features [Indyk]

- Single pass over the data: $a_1, a_2, \dots, a_n$
  - Typically n is known
- Bounded storage (typically n^α or $\log^c n$ or only c)
  - Units of storage: bits, words, or elements (such as points, nodes/edges, …)
  - Impossible to store the complete data
- Fast processing time per element
  - Randomness is OK (in fact, almost necessary)
  - Often sub-linear time for the whole data (skip some)
  - Often approximation of the result

Data stream models classification

- Input stream $a_1, a_2, \dots, a_n$
  - arrives sequentially, item by item
  - describes an underlying signal A,
    signal is a 1D function A: $[1..N] \rightarrow R$
- Models differ on how the input a_i's describe the signal A for increasing i
  (in increasing order of generality):
  a) Time series model - a_i equals to signal $A[i]$
  b) Cash register model- a_i are increments to $A[j]$, $I_i > 0$
  c) Turnstile model - a_i are updates to $A[j]$, $U_i \in R$

a) Time series model (*časová řada*)



- Stream elements a_i are equal to $A[i]$ (a_i's are samples of the signal)
- a_i's appear in increasing order of i (i~time)
- Applications
  - Observation of the traffic on IP address each 5 minutes
  - NASDAQ volume of trades per minute

b) Cash register model (*pokladna*)

+ only

- a_i are increments to signal $A[j]'s$
- Stream elements $a_i = (j, I_i)$, $I_i \geq 0$ to mean

I_i= Increment

$$A_i[j] = A_{i-1}[j] + I_i$$

where $(i \sim \text{time}, \text{j} \sim \text{bucket})$

  - $A_i[j]$ is the state of the signal after seeing i-th item
  - multiple a_i can increment given $A[j]$ over time
- A most popular data stream model
  - IP addresses accessing web server (histogram)
  - Source IP addresses sending packets over a link
  - access many times, send many packets,…

c) Turnstile model (*turniket*)

- a_i are updates to signal $A[j]'s$
- Stream elements $a_i = (j, U_i)$, $U_i \in R$ to mean

+ –

U_i= Update

$$A_i[j] = A_{i-1}[j] + U_i$$

where $(i\sim\text{time, j}\sim\text{bucket, turnstile})$

  - A_i is the state of the signal after seeing i-th item
  - U_i may be positive or negative
  - multiple a_i can update given $A[j]$ over time
- A most general data stream model
  - Passengers in NY subway arriving and departing
  - Useful for completely dynamic tasks
  - Hard to get reasonable solution in this model

c) Turnstile model variants (for completeness)

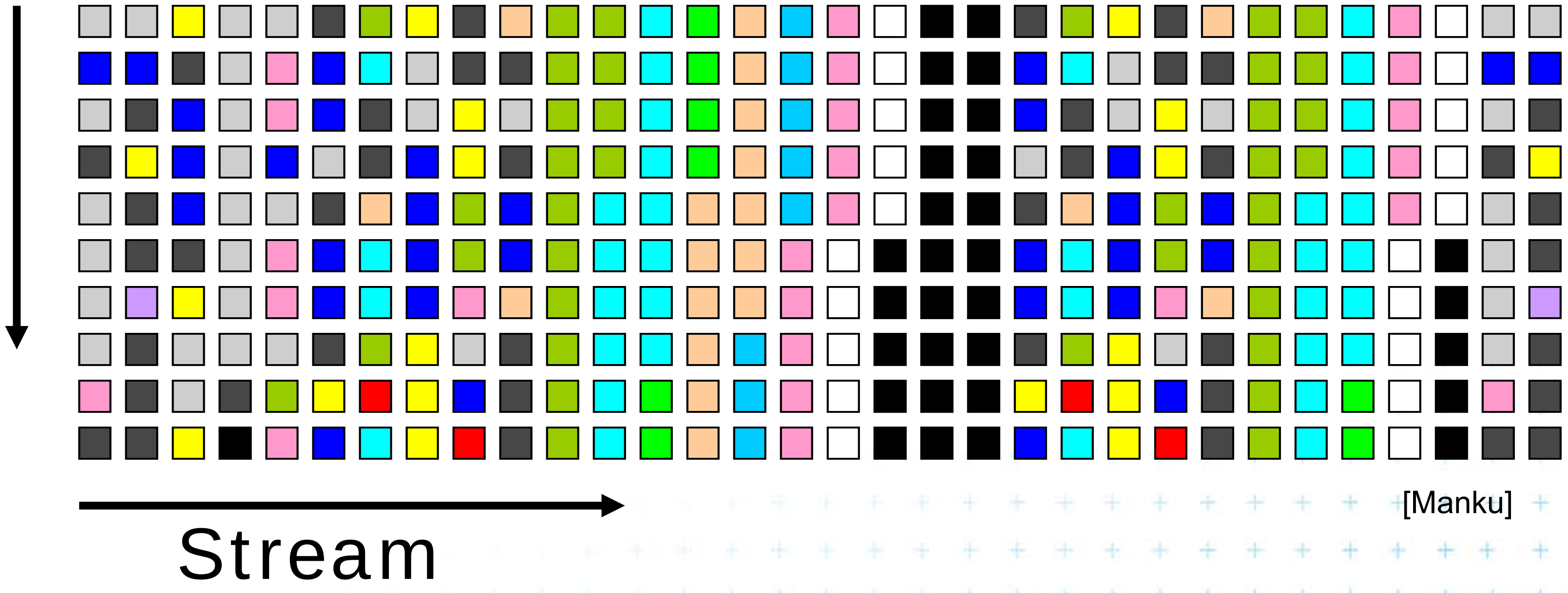
- strict turnstile model – $A_i[j] \geq 0$ for all i Store (*sklad*)
  - The signal A never drops below zero
  - People can only exit via the turnstile they entered in
  - Databases – delete only a record you inserted
  - Storage – you can take items only if they are there
- non-strict turnstile model – $A_i[j] < 0$ for some i Metro
  - Difference between two cash register streams
  - ($A_i[j] < 0$ … negative amount of items for some i)

Examples: Iceberg queries

[Manku]

- Identify all elements whose current frequency f exceeds given threshold $s = 0.1\%$

$$f \geq sN$$



[Manku]

Ex: Iceberg queries – a) ordinary solution

The ordinary solution (not data stream) in two passes

1. Pass – identify frequencies (count the hashes)
   - a set of counters is maintained. Each incoming item is hashed onto a counter, which is incremented.
   - These counters are then compressed into a bitmap, with a 1 denoting a large counter value.
2. Pass – count exact values for large counters only
   - exact frequencies counters for only those elements which hash to a value whose corresponding bitmap value is 1

- Hard to modify for (single pass) data stream – unknown frequencies after only one pass

Ex: Iceberg queries – data stream definition

Input: threshold $s \in (0,1)$, error $\varepsilon \in (0,1)$, $\varepsilon \ll s$, stream length N

Output: list of items and frequencies

- Guarantees:
  - No item omitted (reported all items with frequency > sN)
  - No item added (no item with frequency < $(s - \varepsilon)N$)
  - Estimated frequencies are not less than εN of the true frequencies
- Ex: $s = 0.1\%, \varepsilon = 0.01\%,\ \varepsilon$ should be $\sim \frac{1}{10}$ to $\frac{1}{20}$ of s
  - All elements with freq. > 0.1% of N will output
  - None of element with freq. < 0.09% of N will output
  - Some elements between 0.09% and 0.1% will output

Ex: Iceberg queries – b) sticky sampling

- Probabilistic algorithm, given threshold s, error ε, and probability of failure δ
  - Data structure S of entries (e, f), // S =subset of counters
    e element, f estimated frequency,
    r sampling rate, sampling probability $\frac{1}{r}$
- $S \leftarrow \emptyset, r \leftarrow 1$
- If $e \in S$ then $(e, f{+}{+})$ //count, if the counter exists
  else insert (e, f) into S with probability $\frac{1}{r}$
- S sweeps along the stream as a magnet, attracting all elements which already have an entry in S

Ex: Iceberg queries – b) sticky sampling

S = subset of counters, $|S| < 2t$

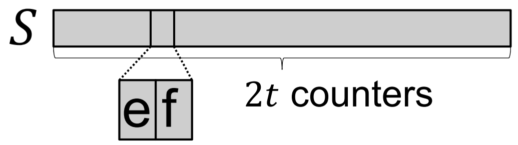
s = threshold

ε = error

δ = probability of failure

- data are processed in blocks
- $t = \frac{1}{\varepsilon}\log\left(\frac{1}{s\delta}\right),$
- size of structure with counters $|S| < 2t$

  $\Rightarrow$ space complexity $|S|$ is independent on N



- r changes over the stream
  - $2t$ elements $r = 1$
  - next $2t$ elements $r = 2$
  - next $4t$ elements $r = 4$ ...

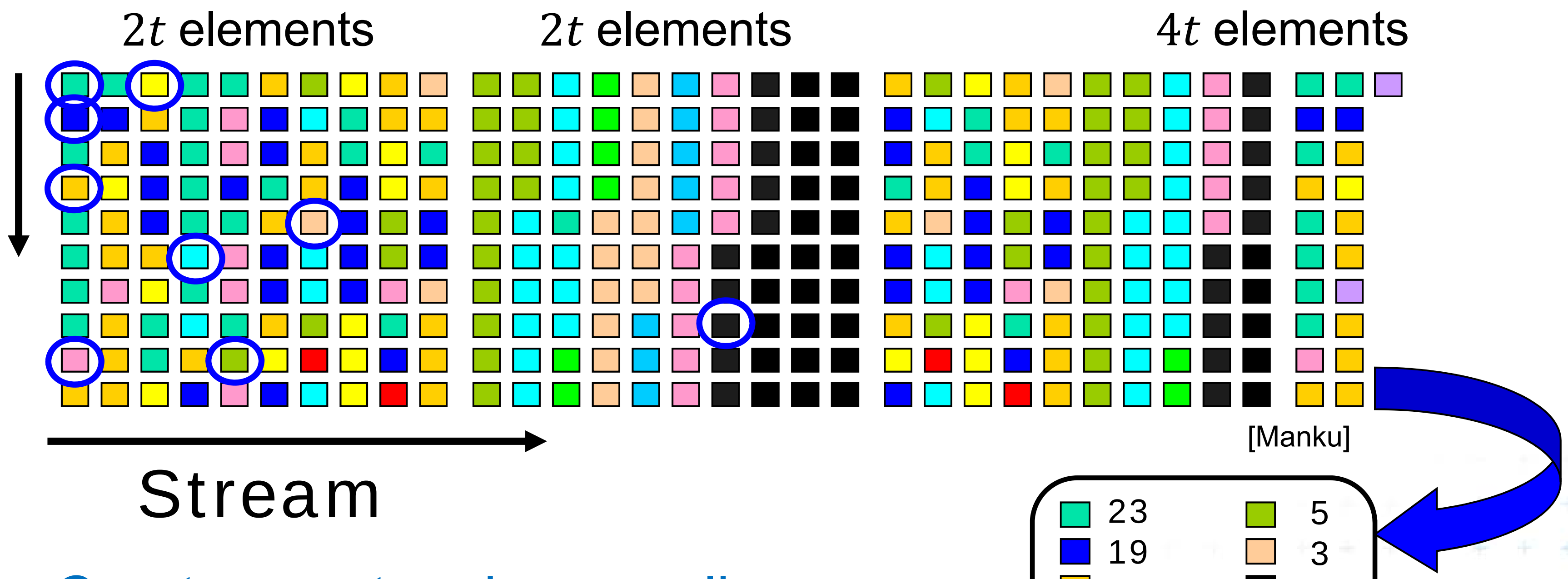
Ex: Iceberg queries – b) sticky sampling

- whenever r changes, we update S
  - only some counters survive
  - for each entry (e, f) in S // random decrement of counters
    - toss a coin until successful (head) // with probability $1/2$
    - if not successful (tail), decrement f
    - if f becomes 0, remove entry (e, f) from S
- Output: list of items with threshold s
  i.e. all entries in S where $f \geq (s - \varepsilon)N$

Ex: Iceberg queries – b) sticky sampling

- Example for
  - support threshold $s = 0.1\%$,
  - error $\varepsilon = 0.01\%$,
  - and probability of failure $\delta = 1\%$
- Sticky sampling computes results
  - with $(1 - \delta) = 99\%$ probability
  - using at most 2t = 80 000 entries
  - $t = \frac{1}{\varepsilon}\log\left(\frac{1}{s\delta}\right) = 40\,000, |S| < 2t$

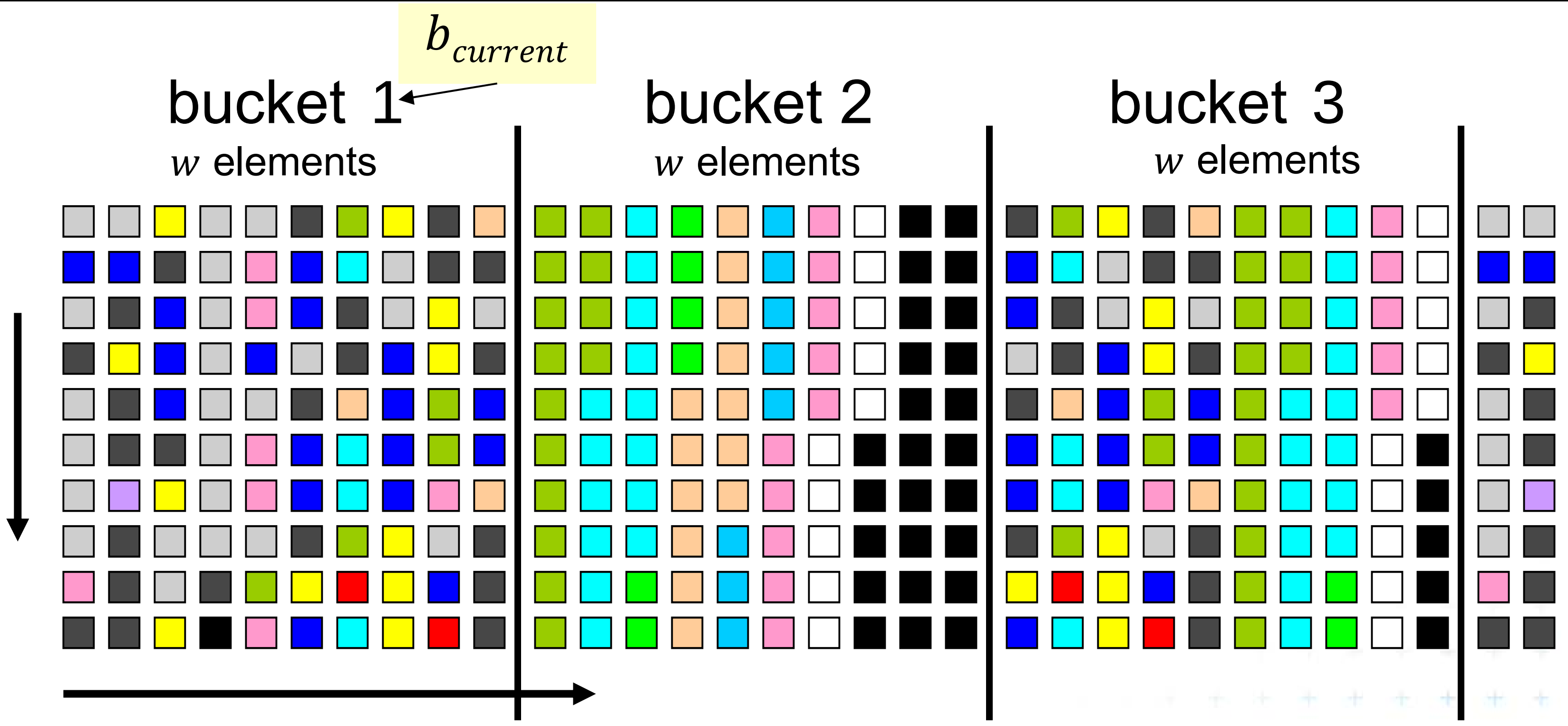
Ex: Iceberg queries – b) sticky sampling



🟩	23	🟩	5
🟦	19	🟧	3
🟨	23	⬛	30
🟪	8		
🟨	11		
🟦	6		

- Create counters by sampling (mind the order of counters)
  - First eight counters created in 1st block
  - Then some counters decremented and $r = 2$
  - Black counter in the second block (with probability $\frac{1}{r} = \frac{1}{2}$)

Ex: Iceberg queries – c) lossy counting



[Manku]

- Divide the stream into buckets
- Keep exact counters for items in the buckets
- Prune entries at bucket boundaries (remove entries for which $f + \Delta \leq b_{current}$)

Ex: Iceberg queries – c) lossy counting

- Deterministic algorithm (user specifies error ε and threshold s)
- Stream conceptually divided into buckets
  - With bucket size $w = \lceil 1/\varepsilon \rceil$ items each
  - Numbered from 1, current bucket id is $b_{current}$
- Data structure D of entries (e, f, Δ),
  - e element,
  - f estimated frequency,
  - Δ maximum possible error of f, $\Delta = b_{current} - 1$ (max number of occurrences in the previous buckets)
- At most $\frac{1}{\varepsilon}\log(\varepsilon N)$ entries

Ex: Iceberg queries – c) lossy counting alg.

- $D \leftarrow \emptyset$ // data structure D of entries (e, f, Δ)
- New element e
  - If $e \in D$ then increment its f
  - If $e \notin D$ then
    - Create a new entry $(e, 1, b_{current} - 1)$
    - If on the bucket border, i.e., $N \; mod \; w = 0$ then delete entries with $f + \Delta \leq b_{current}$
    - i.e., with zero or one occurrence in each of the previous buckets
  - New $\Delta = b_{current} - 1$ is maximum number of times e could have occurred in the first $b_{current} - 1$ buckets
- Output: list of items with threshold s i.e., all entries in S where $f \geq (s - \varepsilon)N$

Comparison of sticky and lossy sampling

- Sticky sampling performs worse
  - Tendency to remember every unique element
  - The worst case is for sequence without duplicates
- Lossy counting
  - Is good in pruning low frequency elements quickly
  - Worst case for pathological sequence which never occurs in reality

Number of mutually different entries 1/2

- Input: stream $a_1, a_2, \ldots, a_n$, with repeated entries
- Output: Estimate of number c of different entries
- Appl: # of different transactions in one day

a) Precise deterministic algorithm:

- Array $b[1..U]$, U = max number of different entries
- Init by $b[i] = 0$ for all i, counter $c = 0$
- for each a_i

  if $b[a_i] = 0$ then $\mathrm{inc}(c)$, $b[i] = 1$ (value has been used)
- Return c as number of different entries in $b[]$
- $O(1)$ update and query times, $O(U)$ memory

Number of mutually different entries

b) Approximate algorithm

- Array $b[1 \ldots \log U]$, $U =$ max number of different entries
- Init by $b[i] = 0$ for all i
- Hash function h: $\{1..U\} \rightarrow \{0..\log U\}$
- For each a_i

  Set $b[h(a_i)]$ = 1 (value has been used)
- Extract probable number of different entries from $b[]$ (many elements hashed to each counter)

Sublinear time example $O(\mathrm{alg}) < O(n)$

- Given mutually different numbers $a_1, a_2, \ldots, a_n$
- Determine any number from upper half of values
- Alg: select k numbers equally randomly
  - Compute their maximum
  - Return this estimation as solution
- Probability of wrong answer = probability of all selected numbers are from the lower half = $\left(\frac{1}{2}\right)^k$
- For error ϵ take $\log\frac{1}{\varepsilon}$ samples
- Not useful for MIN, MAX selection

Randomized algorithms

Randomized algorithms

Motivation

- Array of elements, half of char ”a”, half of char ”b”, Find ”a”
- Deterministic alg:
  - $n/2$ steps of sequential search (when all ”b” are first)
- Randomized:
  - Try random indices
  - Probability of finding ”a” soon is high regardless of the order of characters in the array
    (Las Vegas algorithm – keep trying up to $n/2$ steps)

Why to use randomized algorithms

- May be simpler even if the same worst time
- Deterministic algorithm
  - is not known (prime numbers)
  - does not exist
- Randomization
  - can improve the average running time (with the same worst case time), while
  - the worst time depends on our luck – not on the data distribution
    (It is “hard” to prepare killing datasets)

Randomized algorithms

a) Incremental algorithms
(insert something in random order)
- Linear programming (random plane insertion)
- Convex hulls
- Intersections, space subdivisions (trapezoids)

b) Divide and conquer
(split in random place)
- Random sampling
- Nearest neighbors, trapezoidal subdivisions

Another classification

- Monte Carlo
  - We always get an answer, often not correct
  - Fast solution with risk of an error
  - It is not possible to determine, if the answer is correct
    → run multiple times and compare the results
  - Output can be understand as a random variable
  - Example: prime number test
    - Task: Find $a \in \left\langle 2, \frac{n}{2} \right\rangle$ such as *n* is divisible by a
    - Algorithm: Sample 10 numbers from the given interval, answer
- Las Vegas

Las Vegas algorithms

Las Vegas

- We always get a correct answer
- The run time is random (typically ≤ deterministic time)
- Sometimes fails –> perform restart
- Example: Randomized quicksort
  - No median necessary
  - Simpler algorithm
  - Independent on data distribution
  - Return a correct result
  - The result will be ready in $\theta(n \log n)$ time with a high probability
  - Bad luck – we select the smallest element -> Selection sort

Randomized quicksort (Las Vegas alg.)

RQS(S) = Randomized Quicksort
Input: sequence of data elements, $a_1, a_2, \dots, a_n \in S$
Output: sorted set S

1. Step 1: choose $i \in \langle 1, n \rangle$ in random
2. Step 2: Let A is a multiset $\{a_1, a_2, \dots, a_n\}$
   - if $n = 1$ then output(S)
   - else – create three subsets of $S_<, \; S_=, \; S_>$

$$S_< = \{b \in A : b < a_i\}$$
$$S_= = \{b \in A : b = a_i\}$$
$$S_> = \{b \in A : b > a_i\}$$

3. Step 3: *RQS($S_<$)* and *RQS($S_>$)*

4. Return: *RQS($S_<$)*, $S_=$, *RQS($S_>$)*

Conclusion on randomized algs.

- Randomized algorithms are often experimental
- We would not get perfect results, but nicely good
- We use randomized algorithm if we do not know how to proceed

References

[Kolingerová] Nové směry v algoritmizaci a výpočetní geometrii (1 a 2), přednáška z předmětu Aplikovaná výpočetní geometrie, MFF UK, 2008

[Brönnimann] Hervé Brönnimann. Towards Space-Efficient Geometric Algorithms, Polytechnic university, Brooklyn, NY,USA, ICCSA04, Italy, 2004

[BrönnimannC]Hervé Brönnimann, et al. 2002. In-Place Planar Convex Hull Algorithms. In *Proceedings of the 5th Latin American Symposium on Theoretical Informatics* (LATIN '02), Sergio Rajsbaum (Ed.). Springer-Verlag, London, UK, UK, 494-507. http://dl.acm.org/citation.cfm?id=690520

[Indyk] Piotr Indyk. 6.895: Sketching, Streaming and Sub-linear Space Algorithms, MIT course

[Muthukrishnan] Data streams: Algorithms and applications, (“adorisms” in Google)

[Mulmuley] Ketan Mulmuley. Computational Geometry. An Introduction Through Randomized Algorithms. Prentice Hall, NJ,1994

[Manku] G.S. Manku, R. Motwani. Approximate Frequency Counts over Data Streams, Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002. http://www.vldb.org/conf/2002/S10P03.pdf