Cours ENSL:
Big Data – Streaming, Sketching, Compression

Olivier Beaumont, Inria Bordeaux Sud-Ouest
Olivier.Beaumont@inria.fr
Introduction
Positioning: Memory Aware Complexity of Algorithms

- w.r.t. traditional courses on algorithms
  - Exact algorithms for polynomial problems
  - Approximation algorithms for NP-Complete problems
  - Potentially exponential algorithms for difficult problems (going through an ILP for example)

- Here, we will consider extreme contexts
  - not enough space to transmit input data (sketching) or
  - not enough space to store the data stream (streaming)
  - not enough time to use an algorithm other than a linear complexity one

- Compared to the more "classical" context of algorithms:
  - we aim at solving simple problems and
  - we are looking for approximate solutions only because we have very strong time or space constraints.

- Disclaimer: it is not my research topic, but I like to look at the sketching/streaming papers and I am happy to teach it to you!
Application Context 1: Internet of Things (IoT)

- Connected objects, which take measurements
- The goal is to aggregate data.
- Processing can be done either locally, or on their way (fog computing), or in a data center (cloud computing).
- We must be very energy efficient
  - because objects are often embedded without power supply.
- Energy cost: Communication is the main source of energy consumption, followed by memory movements (from storage), followed by computations (which are inexpensive)
- A good solution is to do as many local computations as possible!
  - but it is known to be difficult (distributed algorithms)
  - especially when the complexity is not linear (e.g. think about quadratic complexity)
- Solution:
  - compress information locally (and on the fly)
  - only send the summaries; summaries must contain enough information!
Application Context 2: Datacenters

- Aggregate construction
- except the network (we can have several levels + infiniband), everything is "linear"
- the distance between certain nodes/data is very large but a strong proximity with certain data stored on disk
- with 1,000 nodes with 1TB of disk and a link at 400 MB/s, we have 1 PB and 400 GB/s (higher than with a HPC system)
- provided the data is loaded locally!
- for 25 TF/s ($10^3 \times 25$ GFs seti@home) in total, ratio 60 (HPC system 40 000)
- in practice, dedicated to linear algorithms and very inefficient for other classes.
- In both contexts, there is a strong need to have data driven algorithms (where placement is imposed by data) whose complexity is linear
Outline of the lectures

• Keywords:
  • Compression, Hashing, Randomized Approximation Algorithms

1. Lecture 1: Two basic theoretical problems
  • Lecture 2: with known lower and upper + randomized and deterministic bounds

2. Lecture 3: Big Data example: Plagiarism detection
  • randomized algorithm + Locality Sensitive Hashing

3. Lecture 4: Randomized Linear Algebra
  • compression beyond Singular Value Decompositions for very large matrices

• Shared Problems
  • Not enough space to store input data
  • Not enough space/time to implement something else than low (linear) complexity algorithms
  • Need for very cheap (online) but dedicated compression algorithm
Sketching – Streaming
• large volume of data generated in a distributed way
  • to be processed locally and compressed before transmission.

• Types of compression?
  • lossless compression
  • compression with losses
  • compression with losses, but tightly controlled loss for a specific function (sketching)

• + we are going to do online (on the fly) compression (streaming)
On-the-fly compression dedicated to a function $f$

- Let $X$ be a stream of numbers (temperatures from a sensor)
- Easy problems?
  - examples: min, max, $\sum$, mean value median?
  - Constraint: compress data and linearize computations
- How?
  - The solution is often to switch to randomized approximation algorithms.
Compression associated to a specific function $f$

- More formally, given $f$ and a stream $X$,
- we want to compress the data $X$ but still be able to compute $\approx f(X)$.
- Sketching: we are looking for $C_f$ and $g$ such that
  - the storage space $C_f(X)$ is small (compression)
  - from $f(X)$, we can recover $f(X)$, i.e., $g(C_f(X)) \approx f(X)$
- Streaming: additional difficulty, the update is performed on the fly.
  - we cannot compute $C_f(\{X, y\})$ from $\{X, y\}$
  - because we cannot store $\{X, y\}$
  - so we need another function $h$ such that $h(C_f(X), \{y\}) = C_f(\{X, y\})$
- and one last difficulty:
  - very often, it is impossible to do in deterministic and exact / deterministic and approximate
  - but only with a randomized and approximation algorithm.
- How to write this?
  - We are looking for an estimator $Z$ such that for given $\alpha$ and $\epsilon$
    - $Pr(|Z - f(X)| \geq \epsilon f(X)) \leq \alpha$. How to read this?
      - the probability of making a mistake by a ratio greater than $\epsilon$ (as small as you want)
      - is smaller than $\alpha$ (as small as you want)
Count the number of visits
Example: count the number of visits / packets

- **Context**
  - a sensor/router sees packets / visits passing through,.....
  - you just want to maintain elementary statistics (number of visits, number of visits over the last 1 hour, standard deviations)
  - Here, we simply want to count the number of visits

- **What storage is necessary if we have** $n$ **visits?** log $n$ bits. Why ?
  Pigeonhole principle. If we have strictly less than log $n$ bits, then we have two events (among the $n$) that will be coded in the same way.

- **What happens if we only allow an approximate answer (say, to a factor of $\rho < 2$)?** you need at least log log $n$ bits. Why ? sketch of the proof: if we use $t < \log \log n$ bits, then we will be able to distinguish less than log $n$ different groups and you can estimate how many groups are needed to count \{0\}, \{0, 1\}, \{0, 1, 2\}, \{0, 1, ..., 7\}.

- **We will look for a randomized and approximated solution**
  - Let us set $\alpha$ and $\epsilon$
  - we are looking for an algorithm that computes $\tilde{n}$, an approximation of $n$
  - that only uses $K \log \log n$ bits storage
  - and such that $Pr(|\tilde{n} - n| \geq \epsilon n) \leq \alpha$
  - $K$ must be a constant...not necessarily a small constant for now!
Crash Course in probabilities

- $Z$ random variable with positive values
- $E(Z)$ is the expectation of $Z$
- definitions and properties?
  - $E(Z) = \int \lambda P(Z = \lambda) d\lambda$ or $E(Z) = \sum_j jP(Z = j)$
  - $E(Z) = \int P(Z \geq \lambda) d\lambda$ or $E(Z) = \sum_j P(Z \geq j)$
  - $E(aX + bY) = aE(X) + bE(Y)$
  - total probabilities (with conditioning) $E(Z) = \sum_j E(Z|Y = j)P(Y = j)$
- To measure the distance from $Z$ to $E(Z)$, we use the variance $V(Z)$
  - Definition?
  - $V(Z) = E((Z - E(Z))^2) = E(Z^2) - E(Z)^2$
  - Properties:
    - $V(aZ) = a^2V(Z)$
    - In general, $V(X + Y) \neq V(X) + V(Y)$ (but it is true if $X$ and $Y$ are independent random variables)
- How to measure the difference between $Z$ to $E(Z)$?
  1. Markov: $Pr(Z \geq \lambda) \leq E(Z)/\lambda$
  2. Chebyshev: $Pr(|Z - E(Z)| \geq \lambda E(Z)) \leq \frac{V(Z)}{\lambda^2E(Z)^2}$
  3. Chernoff: If $Z_1, \ldots, Z_n$ are Independent Bernouilli rv with $p_i \in [0,1]$ and $Z = \sum Z_i$, then
    $Pr(|Z - E(Z)| \geq \lambda E(Z)) \leq 2 \exp(-\frac{\lambda^2E(Z)}{3}).$
Morris Algorithm: Counting the number of events

- Step 1: Find an estimator $Z$
  - $Z$ must be small (of order of log log $n$)
  - we need to define an additional function $g$
  - such that $E(g(Z)) = n$

- Morris algorithm
  - $Z \rightarrow 0$
  - At each event, $Z \rightarrow Z + 1$ with probability $1/2^Z$
  - When queried, return $g(Z) = 2^Z - 1$

- What is the space complexity to implement Morris’ algorithm?
- What is the time complexity in the worst case? What is the expected complexity of a step?
- Prove the correctness: $E(2^Z - 1) = n$ (note $Z_n$ the random variable that denotes $Z$ after $n$ events) Hint: by induction, assuming that $E(2^Z_n) = n + 1$ and showing that $E(2^{Z_{n+1}}) = n + 2$

- How to find a probabilistic guarantee of the type $Pr(|f(Z_n) = \tilde{n} - n| \geq \epsilon n) \leq \alpha$? Hint Prove $E(2^{2Z_n}) = 3/2n^2 + 3/2n + 1$.

- Conclusion? Is this unexpected?
From Morris to Morris+ and Morris+++

- 2nd step: How to get a useful bound?
- Objective: to reduce the variance (the expectation is already what we want). How to do it?
  - Classic idea: do the same experience many times and average them
- Morris algorithm +
  - Morris is used to compute independent $Z_n^{(1)}, Z_n^{(2)}, \ldots, Z_n^{(K)}$
  - On demand, compute $Y_n = \frac{\sum_{i=1}^{K} (2Z_n^{(i)})}{K} - 1$
- Questions:
  - Which space complexity to implement Morris+’s algorithm?
  - What time complexity?
  - Establish the correctness: $E(2Y_n - 1) = n$
  - What is the new guarantee obtained with Chebyshev? How many counters should be maintained?
- How can we do even better?
  - Morris+++ = Morris+(1/3) and median
  - proof with Chernoff: If $Z_1, \ldots, Z_n$ are Independent Bernoulli rv with $p_i \in [0,1]$ and $Z = \sum Z_i$, then $Pr(|Z - E(Z)| \geq \lambda E(Z)) \leq 2 \exp(-\frac{\lambda^2 E(Z)}{3})$. 
How to count the number of unique visitors
2nd example: how to count the number of unique visitors

Context

- It is assumed that visitors are identified by their address ($i_k \in [1, n]$)
- We observe a flow of $m$ visits $i_1, \ldots, i_m$ with $i_k \in [1, n]$ 
- How many different visitors?
- Deterministic and trivial algorithms:
  - if $n$ is small, if $n$ is big... and in front of what?
  - solution in $n:n$ bit array
  - solution in $m \log n$: we keep the whole stream!
- We will see a bit later
  - that we cannot do better with exact and deterministic algorithms
  - that we cannot do better with approximated and deterministic algorithms
- How to do if you cannot store $n$ bits
  - but only $O(\log^k n)$ for a certain $k$?
- we will see that it is again possible by using both randomization and approximation.
- and that no deterministic exact or deterministic approximation can do it with this space constraint.
We will start with an idealized algorithm (which cannot be implemented in practice).

- Let us choose a random $h$ function from $[1, n]$ to $[0, 1]$
- Why idealized?
  - Problem 1: to store such a random function, you must define the images for in each of the $n$ points... at least $\Omega(n)$ bits
  - Problem 2: and in addition we would have to store real values!
  - We will come back to these two problems in a moment....
  - Let us assume for now that storing such a function costs $\Theta(1)$
- How do you keep track of the number of unique visitors?
- We will keep $Z \rightarrow \min_{i \in \text{stream}} h(i)$. Intuition?
  - If you see the same visitor $k$ times, it won’t change $Z$
  - If we see $t$ different visitors, then the values taken by $h$ split $[0, 1]$ in $t + 1$ intervals...and all should have the same size in expectation... and this size is $\frac{1}{t+1}$ including the first !
- so you should return $\frac{1}{Z} - 1$!
Proof of correctness

- Let’s prove that \( E(Z) = \frac{1}{t+1} \).
- \( E(Z) = \int_0^{+\infty} P(Z \geq \lambda) d\lambda \).
  - Show that \( E(Z) = \frac{1}{t+1} \)
  - How to continue? by calculating the variance and applying Chebychev
  - Prove that \( E(Z^2) = \frac{2}{(t+1)(t+2)} \)
  - There is still one foolishness not to be said…. \( E(1/Z) \neq 1/E(Z) \)
  - Intuition: if we can control closely \( Z \) and \( \frac{1}{t+1} \), \( 1/Z - 1 \) will be close to \( t \)
- \( \text{FM}^+ \)
  - Let us maintain \( q = \frac{1}{\epsilon^2 \eta} \) FM instances.
  - \( Z_i \) is the value produced by \( \text{FM}_i \)
  - What to return? \( Y = \frac{1}{(\sum_1^q Z_i)/q} - 1 \)
  - \( E(\frac{\sum_1^q Z_i}{q}) = \frac{1}{t+1} \)
  - \( V(\frac{\sum_1^q Z_i}{q}) = \frac{t}{q(t+1)^2(t+2)} < \frac{E(Z)^2}{q} \)
  - Claim 1: \( P(lY - \frac{1}{t+1} I \geq \epsilon (t+1)) \leq \eta \)
  - Claim 2: \( P(l \frac{1}{Y} - 1 - tl \geq \Theta(\epsilon)t) \leq \eta \)
- \( \text{FM}^{++} \)
  - choose \( \eta = \frac{1}{3} \) adapt \( \epsilon \), instantiate \( K \) copies of \( Y, Y_1, \ldots, Y_K \)
  - output median\( \{ \frac{1}{Y_i} \} \) Ok for \( K = \lceil 36 \log(\frac{1}{\delta}) \rceil \)
Toward a Non Idealized Version. A crucial tool: hashing functions

- We used the set of all possible functions (too large set, too large storage for one function)
- To make it practical, we will consider a large (not too large) family of functions $\mathcal{H}$ from $[1, p] \rightarrow [1, p]$
- How to define the quality of a family $\mathcal{H}$?
- Notion of $k$-wise independence
  - $\forall i_1, \ldots, i_k, \forall j_1, \ldots, j_k, i_k \neq i_l$, and if we pick a random $h$ function in $\mathcal{H}$, then
  - $P(h(i_1) = j_1$ and $h(i_k) = j_k) = 1/p^k$
  - a larger $k$ provides a "better" family
- Examples:
  1. the set of all functions from $[1, p] \rightarrow [1, p]$ is Ok.
     - What $k$, what storage cost?
     - $f(1) \rightarrow p$ choices,..., $f(p) \rightarrow p$ choices
     - Problem: expensive, $p \log p$ bits are necessary for one function
  2. with the polynomials $\mathcal{H}_\text{poly}^k$ of degree $k - 1$ in $F_p$
     - evaluation cost? for degree $k$, $k$ mult & and adds
     - independence? how many polynomials such that $(h(i_1) = j_1$ and $h(i_k) = j_k$
     - exactly one, Lagrange polynomial: $P = \sum_{r=1}^{k} \frac{\prod_{l \neq r} (X - i_l)}{\prod_{l \neq r} (i_r - i_l)} \times j_r$
     - choice? picking a function at random in $\mathcal{H}_\text{poly}^k \rightarrow$ choose $k$ coefficients.
     - and thus the family $\mathcal{H}_\text{poly}^k$ is $k$–independent
Why do we need randomization and approximation?

- Because a deterministic algorithm needs at least $\Omega(n)$ bits
- How to prove this? We assume $n = \Theta(m)$
  - Let us consider the state of the memory of the algorithm after seeing $i_1, \ldots, i_m$
  - We need to prove that there is enough information in what is stored so as to differentiate $2^n$ distinct elements
  - Remark: you can add as many computations as you want!
- Input $X$, let us denote by $C_f(X)$ the state on the memory
- What can be computed using $C_f(X)$ (and only $C_f(X)$)?
  - we can compute $h(C_f(X))$ and $h(C_f(X), \{y\}) = C_f(X \cup \{y\})$
  - do it for all possible $y$ values (visitors)...
  - If $y$ was in the stream, then $h(C_f(X), \{y\}) = h(C_f(X))$ otherwise $h(C_f(X), \{y\}) = h(C_f(X)) + 1$
- In $C_f(X)$, there is enough information to distinguish $2^n$ possible vectors (all visitors vectors)
  - and thus $n$ bits are needed!
Why do we need randomization and approximation?

- Because a deterministic approximation algorithm (say 1.1-approx) needs at least \( \Omega(n) \) bits
- Let us suppose that there exists a collection \( C \) of subsets of \( n \) such that
  - \( |C| \) is large (\( \geq \exp(n/10^4) \))
  - \( \forall S \in C, |S| = n/100 \) (sets are large)
  - \( \forall S_1, S_2 \in C^2, |S_1 \cap S_2| \leq n/2000 \) (intersections are small)
- General idea
  - Let us assume that we have presented to the algorithm one of the sequences of \( C \)
  - Then, we can find back which one!
  - just by trying exhaustively all \( \#C \) sequences with \( C_f(X) \)
  - Since we know how to differentiate exponentially many \( (\exp(n/10^4)) \) elements, we need \( \Omega(n) \) bits
- We still need to prove that such a set \( C \) exists!
  - \( n \) visitors numbered from 1 to \( n \) split into \( n/100 \) packets of 100 visitors
  - In \( S_i \), \( \forall i \) we randomly choose one visitor per packet
  - we build \( \exp(n/10^4) \) such sets \( S_i \).
  - easy: What is their size? \( n/100 \)
  - we need to check that \( \forall i, j, i \neq j, |S_i \cap S_j| \leq n/2000 \)
  - How to do this ?it is enough to prove that the \( \Pr(\text{it works}) \) is \( > 0 \)
  - Why does it work ? \( Y_{i,j} \) number of collisions between \( S_i \) and \( S_j \)
  - \( E(Y_{i,j}) \) ? \( \Pr(Y_{i,j} > n/2000) \) ? \( \Pr(\exists i, j.t.q. Y_{i,j} > n/2000) \) ?
• Step 1: find a $O(1)$-approximation $\tilde{t}$ of $t$ in $O(\log n)$ bits, i.e., a constant $C$ such that $\frac{t}{C} \leq \tilde{t} \leq Ct$ with constant probability (say $\frac{2}{3}$) this is the subject of your homework!
• Playing with constants, let us assume that Step1 provides a 32-approximation with probability $\frac{2}{3}$, then perform $K$ experiments and take the median to have 32-approx with large probability
• To obtain a stronger approximation, we rely on the following technique
• let us chose $g$ in a 2 wise family from $[n]$ to $[n]$.
  1. Imagine that we consider $\log n$ sets, with $S_j$ contains the elements $i$ of the stream s.t. $\text{lsb}(g(i)) = j$.
  2. we know $\tilde{t}$ (close to $t$), let us denote by $Z$ the size of $S_j$ when $2^{j+1} \approx \tilde{t}\epsilon^2$
  3. and let consider $U = 2^{j+1}Z$ in this case
• $E(U) = 2^{j+1}E(Z) = t$, $V(U_i) = 2^{2j+2} \text{Var}(Z) \leq t2^{j+1}$
• so that (Chebychev) $P(|U - t| \geq \epsilon t) \leq \frac{t2^{j+1}}{\epsilon^2 t^2} = \frac{2^{j+1} \tilde{t}}{\epsilon^2 \tilde{t}} \leq C'$
• Then, we use several hashing functions and take the average value to obtain an error with arbitrarily small probability
• Not completely finished ! Is this algorithm implementable this time with small space ?
• No, because $S_0$ is very large for instance ! But the maximum value we are expecting in "interesting" $S_j$ is $\frac{t}{2^{j+1}} = \frac{\tilde{t}}{2^{j+1}} \frac{t}{\tilde{t}} \leq \frac{C}{\epsilon^2}$
• Thus, we can "only" remember the first $\frac{C}{\epsilon^2}$ is each set !
• Overall space complexity ???
Note on Non Idealized FM (3)

- Technique called Geometric sampling
- $n$ elements in the stream, $k \leq n$ distinct elements (with respect to some property)
- Store $\log n$ sub-streams, where $S_0$ stores $1/2$ of the elements (distinct wrt the property), $S_1$ stores $1/4$ of the elements, $S_{\log k}$ stores (close to) 1 element, $S_{\log n}$ a priori stores nothing if $k << n$
- Suppose that when there are $l$ elements in one of the sets, we can find a good estimation of $k$ where typically $l$ is of order $\frac{1}{e^2}$
- Then, we bound all the sets to store less than $10l$ elements (they are useless after that)
- If we have a constant approximation of $k$ (obtained elsewhere), then we know in which set we should look at.
Finding Similar Itemsets
General Idea

• 2 type of difficulties related to
  • the number of objects: $N$ objects $\rightarrow N^2$ comparisons
  • the objects themselves: large texts,…

• Applications
  • pages with a lot of text in common (mirror sites, approximate mirror)
  • plagiarism (today)
  • group news that deal with the same event
  • Amazon, Netflix: users with the same taste
  • dual: Amazon, Netflix: products with the same fans

• we will concentrate on texts, the first step only is application specific
  • Order of magnitude: $10^6$ documents, size a few MB not huge (a few TB)
  • Distributed over a datacenter: large number of nodes $10^3 – 10^6$ nodes

• Two goals:
  • avoid moving data between the nodes (small and shared bandwidth)
  • avoid performing $10^{12}$ comparisons: both for time and data movements
• 3 steps
  1. Shingling: conversion of a large text into a (large) set
  2. Min-hashing: assign to each text a (small) similarity-preserving signature
  3. Locality Sensitive Hashing: detect suspect pairs by collision detection

• randomized approximation algorithm $\rightarrow$ errors: false positive and false negative

• what is crucial in our context? Complexity: we want to deal with linear complexity algorithms only!

• Remark: we assume that the output is (at most) of linear size (otherwise, we have no chance!)
- **k-shingle**
  - sequence of \( k \) successive characters in the text
  - \( \{abcab\} \) et \( k = 2 \)?

- Observation (admitted) close texts \( \rightarrow \) a lot of shingles in common

- A text: represented by the set of shingles it contains

- How many shingles? with \( k = 10 \)

- The data structure should enable to perform comparisons easily...
  - \( 30^{10} \) shingles = \( 2^{49} \)
  - size if stored as a vector of bits 70 TB
  - what happens in practice? Solution? shingles \( \rightarrow \) tokens
  - first use of hashing functions: adapt the size, control collisions
Min-hashing

- Each text is associated to a set of items
- We need to define a similarity between sets.
- Jaccard Similarity: $Sim(C_1, C_2) = \frac{C_1 \cap C_2}{C_1 \cup C_2}$
- $d(C_1, C_2) = 1 - Sim(C_1, C_2)$ is a distance (proof later)
  - one vector per document
  - one row per token
- How to compute the similarity between two documents?
- Problems:
  - we do not want to deal with $N^2$ pairs
  - we cannot centralize all $N$ pairs at a single node
Minhashing

- Let us suppose that \((C_1, C_2)\) are stored at the same place.
- Goal: build a small similarity preserving signature for each document.
- General Idea: build a random game whose expected value (to win) is \(Sim(C_1, C_2)\).
- Do we really need to have \((C_1, C_2)\) at the same place to play the game?
- Do we really need permutations?
- How many hash functions do we need in order to obtain a good precision?
So far: we have a very compact summary of each document (250 × 4B integers= 1kB)

Last step: given a suspicion threshold \( s \in [0, 1] \), return all pairs \((C_1, C_2)\) such that \( \text{Sim}(C_1, C_2) > s \)

Without doing all comparisons!

Order of magnitude:
- \( 10^6 \) documents \(\rightarrow 1\) GO, Ok en mémoire
- \( 10^{12} \) comparisons with \( 10^{-6} \)s per comparison 12 days ;

Goal: go from quadratic to linear complexity

using hash functions again and collision detection

now, we want close vectors to collide, and distant vectors not to collide
split the summary (250 integers) into $b$ blocks of size $r$ ($rb = 250$)

let $h_k$ be the hash function associated to the $k$-th block

collision: (almost) only if both vectors coincide on this block. Solution: use a large number of buckets (with respect to $10^6$) $\rightarrow 10^9$ is Ok in practice, very few false positive.

a pair $(C_1, C_2)$ is suspicious if $\exists k, h_k(B^1_k) = h_k(B^2_k)$ where $B^i_k$ is the $k$-th block of $C_i$

what happens ?
  • if $r$ is too small ? too many false positive
  • if $r$ is too big ? we will miss similar itemsets and get false negative

Given $r$ (and thus $b$) and $s = Sim(C_1, C_2)$, what is the probability that a collision occurs ?
with $N = 100$ and $r = 3, 4, 5$

false positive? false negative?
with $N = 1000$ and $r = 3, 4, 5$
Practical implementation?

- distributed documents.
- keep everything local (until the computation of signatures)
- keep everything local (compute the hashing of each block) 1 document + 1 block $\rightarrow$ 4B !
- gather all information related to one block number to the same node (4B + 1B for the index) $\rightarrow$ 5MB
- detect all suspicious pairs (very few and send them to a specific node)...
- very few communications!
Locality Sensitive Hashing and Nearest Neighbors Search
(\(k\)-) nearest neighbors

- Metric space with distance, set \( P \) of points
- preprocessing allowed on \( P \)
- Query: given a point, find its \((k)\) closest neighbor(s)
  - example for spam classification: start with a huge annotated emails
  - one word = one item
  - return the \( k \) closest emails, majority vote to determine if it is spam or not
- Approach \# 1: no preprocessing, just look through all possible items
  - space \( O(dn) \)
  - query \( O(dn) \)
- Approach \# 2: if \( d=1 \)
  - space \( O(n) \) just keep the boundaries
  - query \( O(\log n) \) just a basic binary search
- Approach \# 3: if \( d=2 \)
  - Voronoï diagrams: space \( O(n) \) and computing cost \( O(n \log n) \)
  - query time easy (locate the cell)
  - As dimension increase, the description increases exponentially with \( d \)

- all exact (known) approaches in high dimension either have
  - exponential space \( O(n^d) \)
  - or exponential query time!
  - (same for \( kd\)-trees)
  - in very large dimension, the naive algorithm is not that bad!
• Given a set of $P$ points, construct a data structure such that
  • on query $q$, we return $p$ in $P$ such that
  • $d(p, q) \leq c \min_{p' \in P} d(p', q)$

• $(r_1, r_2)$ PLEB problem: point location in equal balls
  • given a set $P$ of points and $r_1, r_2$
  • construct a data structure to answer as follows
  • If $\exists p \in P$ st $d(p, q) \leq r_1$, return YES and any $p' \in P$ s.t. $d(p', q) \leq r_2$
  • If there is no $p \in P$ st $d(p, q) \leq r_2$, return NO
  • elif don't care what algorithm returns
Locality Sensitive Hashing (Indyk, Motwani)

- Usually, we want hashing functions to "shuffle" items as much as possible
  - When writing $P(h(i_1) = j_1 & h(i_2) = j_1) = 1/p^2$, we say that the distance between images should not depend on the distance between initial points

- Here, we want to detect "collisions"
  - we want close points to have a high probability to collide
  - we want distant points to have a low probability to collide
  - just as in the context of plagiarism.

- General idea
  - hash items into many different buckets (with different functions)
  - declare that there is a collision if two items fall into the buckets

- Formal definition $\mathcal{H}$ a family of hash function $U \rightarrow S$
- (where $U$ is the set of points, $S$ the set of buckets)
- is said to be $(r_1, r_2, p_1, p_2)$ locality sensitive if
  - If $d(p, q) \leq r_1$, then $P_{h \in \mathcal{H}}(h(p) == h(q)) \geq p_1$ and
  - If $d(p, q) \geq r_2$, then $P_{h \in \mathcal{H}}(h(p) == h(q)) \leq p_2$
- of course $r_1 < r_2$, $p_1 > p_2$
Example

- Let $H^d = \{0, 1\}^d$ equipped with Hamming distance (number of different coordinates)
- Let $\mathcal{H} = \{h_i, \forall i, \text{ where } h_i(b_1, \ldots, b_d) = b_i\}$
- $\mathcal{H}$ is $(r, cr, 1 - r/d, 1 - cr/d)$ locality sensitive
  - if $p, q$ are at distance at most $r$, they have at least $(d - r)$ coordinates in common and thus a probability at least $1 - r/d$ to be hashed similarly,
  - if $p, q$ are at distance at least $cr$, they have at most $(d - cr)$ coordinates in common and thus a probability at most $1 - cr/d$ to be hashed similarly.
Master Theorem of LSH

Theorem
Suppose \( \exists (r_1, r_2, p_1, p_2) \)-LSH family, then there is an algorithm for \((r_1, r_2)\)-PLEB with answer queries with constant probability (it might be wrong), and that uses space \(O(dn + n^{1+\rho})\) and query time \(O(n^\rho)\) (evaluation of hash functions), where \(\rho = \frac{\log(1/p_1)}{\log(1/p_2)}\) (complexity decreases when \(\rho\) decreases, ie when \(p_2 \ll p_1\)).

Sketch of the proof — Algorithm

- let \((k, l)\) be parameters (t.b.d. later), let \(G\) be a family of hash functions from \(U\) to \(S^k\) (new buckets), \(g(p) = (h_{g_1}(p), \ldots, h_{g_k}(p))\) each \(h_{g_i}\) being randomly chosen in \(H\).
- Preprocessing:
  - (1) choose \(g_1, \ldots, g_l\) (other parameter) independently from \(G\)
  - (2) for each \(p \in P\), store \(g_1(p), \ldots, g_l(p)\)
- On query
  - (1) search the points of \(P\) in \(g_1(q), \ldots, g_l(q)\), but stop after the first \(2l\) points in (the unlikely) case there are more than \(2l\).
  - (2) If there is one point \(p\) such that \(d(p, q) \leq r_2\) return it and return YES, otherwise return NO
Proof (continued)

- Intuition (1): if \( q \) and \( p \) are "close", then one the \( g_i \) will send them into the same \( k \)-bin.
- Intuition (2): it is unlikely that they are \( 2l \) distant (useless) points in the set (that would prevent to find the useful point)
- (2) There are at most \( 2l - 1 \) points st \( d(p, q) > r_2 \) and \( \exists j, g_j(p) = g_j(q) \) with constant probability
  - Let \( k = \log_{1/p_2} n \), what is the expected number of points st (2) holds?
  - If \( d(p, q) > r_2 \) then for each \( h \), the probability of collision is at most \( p_2 \)
  - so the expected number of times \( p \) is written in any \( g_j \) is \( p_2^k \), and the expected number of times it is written for a given \( g \) is \( lp_2^k \)
  - and the number of "bad points" written for \( g \) is therefore at most \( nlp_2^k = l \)
  - Markov says \( P(X > \lambda) < E(X)/\lambda \) so \( P(\text{more than } 2l \text{ bad points}) \leq 1/2 \)
- (1) If \( p \in P \) with \( d(p, q) \leq r_1 \), then \( \exists j, g_j(p) = g_j(q) \) with constant probability
  - If \( d(p, q) \leq r_1 \) then for each \( h \), the prob of collision for \( h \) is at least \( p_1 \)
  - the probability of collisions in one bucket is \( p_1^k \)
  - the probability of a collision in at least one of the \( l \) buckets is \( 1 - (1 - p_1^k)^l = 1 - (1 - n^p)^l \)
  - choice of \( l \) ? if we set \( l = n^p \) then the probability is at least \( 1 - 1/e \simeq 0.63 \).
- to increase the probability, use the classical and tricks
- Check space and time complexities

38
Conclusion on sketching/streaming/compression

- **Goal:** data flow $X$ and a function to evaluate $f$
- **streaming:** maintain a summary $C_f(X)$ enough to compute $f(X) \approx g(C_f(X))$
  - Solution: Use approximation randomized algorithms
    - $\forall \epsilon, \delta$, $Pr(\text{relative error} \geq \epsilon) \leq \delta$
  - enough (and often necessary) to change space complexities (from $\log n \to \log \log n$, $n \to \log n$, from $n^d$ to $n^{\rho d}$)
  - at the price of sometimes large constants
  - but constants are pessimistic
  - and very small $\epsilon$ and $\alpha$ are not always required (plagiarism)

- **General Idea:**
  - Do less communications (same a lot of energy, time)
  - But more local computations (cheap)
  - crucial for IoT and datacenters

- **Method:**
  - Find an estimator $Z$ tel que $E(Z) = \text{what we want to estimate}$
  - go for $+$ and $++$ versions to control the probability
  - hash functions are a very powerful and versatile tool:
    - to shuffle potentially correlated entries (Unique Visitors)
    - to adapt the size of sets (Plagiarism)
    - to create short summaries (Min-Hashing)
    - to detect close items (Locality Sensitive Hashing)