CS5234 Algorithms at Scale

$e \approx 2.718$
$e^{-1} \approx 0.368$
$e^{-2} \approx 0.135 < \frac{1}{6}$

- Fact: $e^{-2x} \le 1 - x \le e^{-x}$
- Markov's ineq.: If X is a nonnegative ran. var then $\forall k > 0$: $Pr[X \ge k] \le \frac{E[X]}{k}$
- Chebyshev ineq: If X is a ran. var then $\forall k > 0$: $Pr[|X - E[X]| \ge k] \le \frac{Var[X]}{k^2}$
- Variance: $Var[X] = E[(X - E[X])^2] = E[X^2] - E[X]^2$
  - Where X, Y are indep: $\cdot E[XY] = E[X] \cdot E[Y]$
    $\cdot Var[X+Y] = Var[X] + Var[Y]$
- Hoeffding bound: If $X_1, \dots, X_n$ are indep. ran.vars s.t. $\forall i, X_i \in [a_i, b_i]$,
  then $\forall \delta > 0$, $Pr[|Z - E[Z]| \ge \delta] \le 2\exp\left(-\frac{2\delta^2}{\sum (b_i - a_i)^2}\right)$ where $Z = \sum_{i=1}^{n} X_i$ and $\underbrace{E[Z] = \sum_{i=1}^{n} E[X_i]}_{\text{by linearity of exp.}}$
  - Where $a_i = 0$ and $b_i = 1$, we get: $\forall \delta > 0$, $Pr[|Z - E[Z]| \ge \delta] \le 2\exp\left(-\frac{2\delta^2}{n}\right)$
- Adjlist format graph: get i^{th} neighbour of node u in constant time
- Connectivity problem on sparse graphs (n nodes, m edges, d max. degree) → return true if connected
  → return false if ε-far from connected (i.e. graph cannot be connected even if you can add/remove at most εnd entries in adjlist (note: each edge has two entries in the adjlist))
  - Lemma: If G is ε-far from connected, then it has $\varepsilon dn/4$ connected components
    (note: when add/removing edges, must ensure max degree is satisfied)
  - Lemma: If G is ε-far from connected, then it has $\varepsilon dn/8$ connected components of size $\le 8/\varepsilon d$
  - Algo: repeat $16/\varepsilon d$ times: pick random node and check if component size is less than $8/\varepsilon d$.
    Takes $O(d \cdot (8/\varepsilon d)) = O(\frac{1}{\varepsilon})$ per check, total time $O(\frac{1}{\varepsilon^2 d})$
  - If G is ε-far from connected, each iteration has at least $\frac{\frac{\varepsilon dn}{8}}{n} = \frac{\varepsilon d}{8}$ prob. of finding small component
  - So $P(\text{algo returns true when } \varepsilon\text{-far}) \le (1 - \frac{\varepsilon d}{8})^{\frac{16}{\varepsilon d}} \le e^{-2} \le \frac{1}{3}$
- Multiplicative approximation: $C^*(1-\varepsilon) \le C \le C^*(1+\varepsilon)$
  (C^*: correct answer; C: estimate; C^*: correct answer)

- Approx. connected components algo:

```
sum = 0
for j = 1 to s :
    u ← randVertex()
    if u has at least 2/ε nodes reachable (BFS)
        sum = sum + ε/2
    else (found n(u) nodes)
        sum = sum + 1/n(u)
return n·(sum/s)
```

  - Result: with probability $> \frac{2}{3}$, output is within $CC(G) \pm \varepsilon n$
    takes $O(\frac{d}{\varepsilon^3})$ time (d: max degree)
  - Result: with probability $> 1 - \frac{1}{\delta}$, output is within $CC(G) \pm \varepsilon n$
    takes $O(\frac{d \ln \delta}{\varepsilon^3})$ time

- MST approx. algo: outputs a $(1 \pm \varepsilon)$-multiplicative approximation
  (for integer edge weights in $\{1, \dots, W\}$, assuming graph is connected)

```
sum = n - W
for j = 1 to W-1                      ε' := ε/W , δ := 1/(3W)
    sum += ApproxCC(G_j, d, ε', δ)
return sum
```

  (G_j: graph containing only edges with weight $\le j$)
  - Result: with probability $> \frac{2}{3}$, output is within $MST(G)\ (1 \pm \varepsilon)$
    takes $O\left(\frac{dW^4 \log W}{\varepsilon^3}\right)$ time

- Maximal matching approx algo:

```
query(e):
    for all neighbours e' of e:
        if hash(e') < hash(e):
            if query(e') = true
                return false
    return true
```

→ query(e) is supposed to return true iff e is part of the greedy maximal matching induced by the hashes

- Expected time complexity of query $\le 2 \sum_{k=1}^{\infty} \frac{d^k}{k!} = O(e^d)$
- Algo to approx. maximal matching size:

```
sum = 0
for j = 1 to s :
    choose edge uniformly at random.
    if (query(e)) then sum = sum + 1
return m·(sum/s).
```

- Chernoff bounds: If $X_1, \dots, X_n$ are indep ran.vars s.t. $\forall i, X_i \in [0, s]$, and let $X = \sum_{i=1}^{N} X_i$ and $\mu = E[X]$,
  then: for any $0 \le \delta \le 1$: $\cdot Pr[X \ge (1+\delta)\mu] \le e^{-\frac{\mu\delta^2}{3s}}$
  $\cdot Pr[X \le (1-\delta)\mu] \le e^{-\frac{\mu\delta^2}{2s}}$

- **Yao's minimax principle**: Given a problem, let X be the set of inputs, $\Gamma(X)$ be the set of probability distributions on X, D be the set of deterministic algos, R be the set of randomised algos:

$$\forall A \in R, \ \forall \gamma \in \Gamma(X) \ : \ \max_{x \in X} \underset{\substack{\text{randomness}\\ \text{of } A}}{E} [\text{cost}(A, x)] \geq \min_{B \in D} \underset{x \sim \gamma}{E} [\text{cost}(B, x)]$$

  - To show that the expected cost of any randomised algorithm (on the worst case input) is $\geq T$, it suffices to show that there is an input distribution $\gamma \in \Gamma(X)$ such that for any deterministic algorithm $B \in D$, $\underset{x \sim \gamma}{E}[\text{cost}(B, x)] \geq T$

- **Streaming Algorithms**:
  - only have a small scratch space, but want to calculate some property of the items in the stream.

E.g.:
  - return an approximation of the number of times x appears:
    - $\text{count}(x)$: $N(x) - \varepsilon m \leq \text{count}(x) \leq N(x) + \varepsilon m$ ($N(x)$: real count; m: length of stream)
    - heavy hitter: return every item appearing $\geq 2\varepsilon m$ times but no item appearing $< \varepsilon m$ times

- **Misra-Gries algorithm**:
  - Set P of <item, count> pairs
  - For each u in stream:
    - if $\langle u, c\rangle$ is in P, increment c.
    - else add $\langle u, 1\rangle$ to P.
    - if $|P| > k$, decrement c for every $\langle v, c\rangle$ in P
    - remove from P all $\langle v, c\rangle$ where $c = 0$.
- $\text{count}(x)$: return c if $\langle x, c\rangle$ in P, otherwise return 0.
- **Heavy hitters**: return x if $\text{count}(x) \geq \varepsilon m$.

- **Flajolet-Martin (FM) algorithm**:
  - $x = 1$
  - for each u in the stream:
    - if $h(u) < x$ then $x = h(u)$ (h: hash)
  - return $\frac{1}{x} - 1$

- **FM+ algorithm**:
  - run a copies of FM, to get $X_1, \dots, X_a$ (these are the x from FM)
  - compute $Z = \frac{1}{a} \sum_{j=1}^{a} X_j$
  - return $\frac{1}{Z} - 1$

- **FM++ algorithm**:
  - run b copies of FM+, to get $Y_1, \dots, Y_b$ ($Y_j = \frac{1}{Z_j} - 1$)
  - return the median of Y_i

- **Result**: $a = \frac{4}{\varepsilon^2}$, $b = 36 \ln \frac{2}{\delta}$, then with probability at least $1 - \delta$, FM++ returns an answer in $t(1 \pm 4\varepsilon)$.

- **Streaming a graph**: each edge is an element in the stream, and edges arrive in arbitrary order.
  - UFDS-based:
    - count connected components: $O(n \log n)$ space (since $\log n$ bits to store each vertex) and $O(\alpha(n, n))$ update cost
    - check if graph is bipartite: $O(n \log n)$ space and $O(\alpha(n, n))$ update cost (α: inverse Ackerman function)

- **Shortest path approx.**
  - Find a "spanner": a spanning subgraph $H \subseteq G$ s.t. H is sparse (i.e. not too many edges) and for all $u,v \in V(G)$: $d_G(u,v) \le d_H(u,v) \le \underset{\text{"stretch"}}{\underset{\uparrow}{\alpha}} d_G(u,v)$.
  - note: if every edge $uv \in E(G)$ satisfies $\frac{d_H(u,v)}{d_G(u,v)} \le \alpha$, then $\forall u,v \in V(G)$ (not necessarily an edge), $\frac{d_H(u,v)}{d_G(u,v)} \le \alpha$

  Algo:
  - For each edge uv in stream:
    - if $d_H(u,v) > 2k-1$ then:
      - add uv to H
  - return H

  Thms:
  - The girth of H is $> 2k$ (girth: length of smallest cycle)
  - If $\text{girth}(H) > 2k$ then H has $O(n^{1+\frac{1}{k}})$ edges

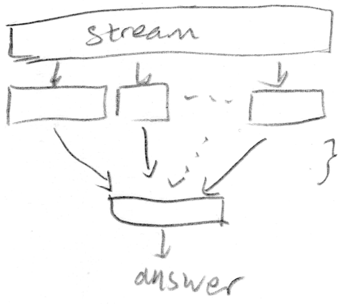
- If we pick $k=2$, then space $= O(n^{\frac{3}{2}} \log n)$
- If we pick $k = \log n$, then space $= O(n^{1+\frac{1}{\log n}} \log n) = O(n \log n)$

- **Matching approx:**
  - Do greedy matching - pick an edge if both vertices are still not matched
  - It is a 2-approximation
- **Weighted matching:** Graph edges have weights, want to find max weight matching

  Algo:
  - M: matching, initially empty.
  - For each edge uv in stream:
    - let C be the set of edges in M that are incident on u or v
    - if $w(uv) > (1+\gamma)w(C)$ then:
      - remove C from M
      - add uv to M
- Result: it is a 6-approximation of optimal.

- **Clustering:**
  - k-centre clustering: choose k points (i.e. centres) that minimise the maximum distance to a centre
  - k-median clustering: choose k points that minimise the average distance to a centre
    - $D(P,C) := \sum_{i=1}^{n} |p_i - c(i)|$ where $P = \langle p_1, \dots, p_n \rangle$ are the points and $C = \langle c_1, \dots, c_k \rangle \subseteq P$ and c is a function mapping a point to a centre
    - C is an (α, γ)-approx: $|C| \le \alpha |C^*|$ and $D(P,C) \le \gamma D(P, C^*)$
    - ILP solution (NP-hard):
      - $y_j := \begin{cases} 1 & \text{if } p_j \text{ is a centre} \\ 0 & \text{otherwise} \end{cases}$
      - $x_{i,j} := \begin{cases} 1 & \text{if } p_i \text{ is assigned to centre } p_j \\ 0 & \text{otherwise} \end{cases}$
      - minimise $\sum_{i,j} x_{i,j} d(p_i, p_j)$ where
        - $\forall i : \sum_j x_{i,j} = 1$
        - $\sum_j y_j \le k$
        - $\forall i,j : x_{i,j} \le y_j$
        - $\forall i,j : x_{i,j}, y_j \in \{0,1\}$
    - Solve the LP version, then do some rounding.

- **Core-Set algorithm for streaming k-median:**
  - $C = \emptyset$
  - repeat $\sqrt{\frac{n}{k}}$ times:
    - Let P = next $\sqrt{nk}$ points
    - Find $(2,4)$-approx clustering on P
    - Add $2k$ new cluster centres to C, but weight each cluster centre with the number of points attached to it
  - return $(2,4)$-approx (weighted) clustering on C.



} core set: set of $2k\sqrt{\frac{n}{k}} = 2\sqrt{nk}$ centres

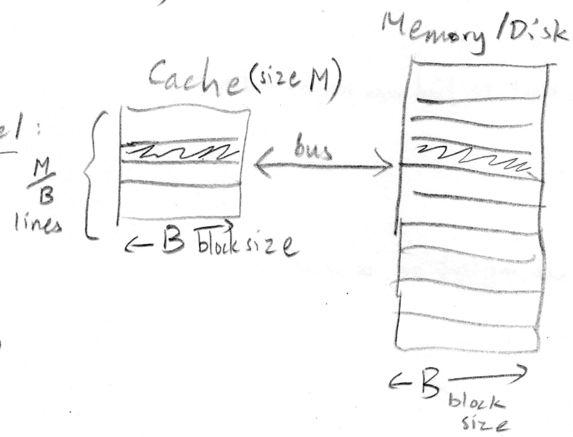
Space: $O(\sqrt{nk})$

output: $(2,80)$-approx of k-median

proof: generally by Δ-ineq.

- can stack more layers if we want a further reduction in space:
  - let $m = n^{\varepsilon}$ (m elements before grouping to the next level)
  - num levels $= \log_m n = \frac{1}{\varepsilon}$
  - space $= \frac{2kn^{\varepsilon}}{\varepsilon}$
  - approx factor $= O(8^{\frac{1}{\varepsilon}})$

- Caching
- External Memory Model:

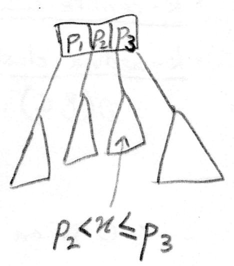


← entire cache line gets copied over when we want to access anything on it

← want to minise the number of times a cache line gets transferred

- Examples:
  - Scanning data:
    - Linked list: $O(N)$
    - Array: $O(N/B)$
  - Searching data:
    - Linked list: $O(N)$
    - Red-black tree: $O(\log N)$
    - (sorted) Array: $O(\log \frac{N}{B})$
    - B-tree: $O\left(\frac{\log N}{\log B}\right)$
  - Sorting data:
    - B-tree: $O(N \log_B N)$
    - Buffer tree: $O\left(\frac{N}{B}\log_{\frac{M}{B}} \frac{N}{B}\right)$

- **(a,b)-trees:**
  - tree structure
  - satisfies search property
  - $b \geq 2a$
  - all keys stored in leaves
  - internal nodes store pivots to guide search
  - root has ≥ 2 children
  - non-root nodes have $\geq a$ children
  - all nodes have $\leq b$ children
  - all leaves have same depth



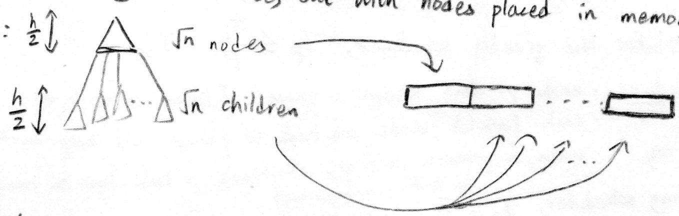
Properties:
- height of tree $\leq \log_a\left(\frac{n}{a}\right) + 1 \in O(\log_B n)$ — n: number of keys; where $a, b \in O(B)$
- insert: insert into correct leaf, then split from bottom up if $> b$ keys
- delete: remove from correct leaf, then either merge or rebalance siblings from bottom up if $< a$ keys

amortized cost (w/ parent ptr):
- per node: $O(1)$
- per operation: $O(\log_B n)$

amortized cost:
- per node: $O(\frac{1}{B})$
- per operation: $O(\frac{1}{B}\log_B n)$

- <u>Buffer tree</u>: (for fast searching and <u>very fast</u> insertion/deletion)
  - Build a (2,4)-tree, but add a buffer of size 2B to every node.
  - For each leaf: ensure that it has between B and 5B keys (inclusive).
  - <u>insert</u>:
    - add ins[key] to root buffer
    - Clean buffer: remove any del[key] or duplicate ins[key]
    - If |buffer| ≥ B, flush the buffer

    $\Big\}$ $O(1)$ + buffer flush $= O(\frac{1}{B}\log n)$
  - <u>delete</u>: similar to insert → $O(1)$ + buffer flush $= O(\frac{1}{B}\log n)$
  - <u>search</u>: walk from root to leaf, remember to search buffer too : $O(\log n)$ ← branching factor is 3, not B.
  - <u>flush</u>:
    - sort buffer
    - move operations to children's buffers ← each item (ins[key]/del[key]) contributes $\Theta(\frac{1}{B})$ to flush operation
    - Clean children's buffers.
    - recursively flush children's buffers if necessary.
  - leaves have no buffer — all the keys are stored there.
    - consider splitting/merging leaves as necessary.
- <u>$\sqrt{B}$ Buffer tree</u>: (each node has $\sqrt{B}$ children)
  - <u>insert/delete</u>: $O(\log_B n)$
  - <u>search</u>: $O\left(\frac{\log_B n}{\sqrt{B}}\right)$ (tree depth: $2\log_B n$)
- <u>van Emde Boas search tree</u>: static <u>cache-oblivious</u> search tree (don't know M and B)
  - Like a normal balanced binary search tree, but with nodes placed in memory in a special way.
  - recursively layout:



  - search: $O(\log_B n)$

<u>Cache-efficient graph algorithms</u>:

- <u>Breadth-first search</u>: Layer by layer: Graph is stored as adjlist format.
  For each layer:
  1. L_{i+1} ← neighbours of nodes in L_i $\quad O(|L_i| + edges(L_i)/B)$
  2. Sort L_{i+1} $\quad O(sort(L_{i+1}))$
  3. Remove duplicates in L_{i+1} $\quad O(edges(L_i)/B)$
  4. $L_{i+1} \leftarrow L_{i+1} \setminus L_i$ $\quad O(|L_i|/B + edges(L_i)/B)$
  5. $L_{i+1} \leftarrow L_{i+1} \setminus L_{i-1}$ $\quad O(|L_{i-1}|/B + edges(L_i)/B)$

  Total cost $= O(|V| + |E|/B + sort(|E|))$

- <u>Count connected components</u>: edges are stored in a single array.
  - some recursive algorithm like UFDS
    $O(sort(E)\log(E))$
- <u>MST</u>: D&C on edge weight:
  - Divide E into E1 and E2.
  - Recursively find MST T1 of E1.
  - Contract E1.
  - Recursively find MST T2 of E2.
  - Expand E2.
  - Return T1 ∪ T2.
  - $O(sort(E)\log(\frac{E}{M}))$

Parallel Algorithms

- **PRAM model:**
  - p processors
  - shared memory
  - processors take one step at each clock tick
  - each processor can be programmed separately

- **Fork/Join:** (relies on a good scheduler)
  - E.g.: Sum(A, b, e): (A = [1,...,n])

```
Sum(A, b, e):
    if b = e:
        return A[b]
    mid = (b+e)/2
    fork:                          (fork)
        L ← Sum(A, b, mid)     } done in parallel
        R ← Sum(A, mid+1, e)   }
    sync                           (join)
    return L+R
```

  - work $= O(n)$
  - span $= O(\log n)$

E.g. AllZero$(A, 1, n, p)_j$:

```
for i := (n/p)·(j-1)+1 to (n/p)·(j):
    if A[i] ≠ 0 then answer ← false
done ← done+1
wait until done = p
return answer
```

- **Parallel merge sort:**
  - work $= O(n\log n)$
  - span $= O(n)$ (naive merge)
  - span $= O(\log^3 n)$ (binary search parallel recursive merge is $O(\log^2 n)$ per merge.)

- **Work:** total steps done on all processors: T_1
- **Span:** longest path in the program: T_∞
- **Parallelism:** $\frac{T_1}{T_\infty}$ ($\approx$ number of processors that we can use productively)
- **On P processors:** want: $T_p \approx \underbrace{\frac{T_1}{P}}_{\text{parallel part}} + \underbrace{T_\infty}_{\text{sequential part}}$

- **Greedy scheduler:**
  - If $\leq p$ tasks are ready, execute all of them
  - If $> p$ tasks are ready, execute any p of them
- **Brent-Graham thm:** For the greedy scheduler, $T_p \leq \frac{T_1}{P} + T_\infty$
- **Work-stealing scheduler:**
  - each process keeps a queue of tasks to work on
  - each fork() adds one task to queue, and keeps working
  - when a process is free (no tasks in own queue), it steals a task from a random queue.
- **Thm:** For work-stealing scheduler, $T_p \leq \frac{T_1}{P} + O(T_\infty)$

Parallel operations on a balanced binary search tree:

- insert/delete/divide: $T_1 = T_\infty = O(\log n)$
- union/subtraction/intersection/difference (set symmetric difference): $T_1 = O(n+m)$, $T_\infty = O(\log n + \log m)$

E.g. Union(T1, T2):

```
Union(T1, T2):
    if T1 = null return T2
    if T2 = null return T1
    key ← root(T1)
    (L, G, x) ← split(T2, key)
    fork:
        TL ← Union(key.left, L)
        TR ← Union(key.right, R)
    sync.
    T ← join(TL, TR)
    insert(T, key)
    return T.
```

Basic building blocks operations for BBST ($O(\log n + \log m)$):

- split$(T, k) \to (T_1, T_2, x)$ (x could be null; x is the item at k if exists)
- join$(T_1, T_2) \to T$ (assumes $\forall x_1 \in T_1, \forall x_2 \in T_2, x_1 < x_2$)
- root$(T) \to x$ (get the root item of T, leaving T unchanged)
- insert$(T, x) \to T'$

- **Parallel BFS using BBST for storage:**

```
F ← {s}
D ← {}
while F ≠ ∅:
    D ← Union(D, F)
    F ← ProcessFrontier(F)     (recursive divide/process/union)
    F ← SetSubtract(F, D)
```

	Work	Span
D ← Union(D,F)	$O(m\log n)$	$O(\log^2 m)$
F ← ProcessFrontier(F)	$O(m\log^2 n)$	$O(\log^3 m)$
F ← SetSubtract(F,D)	$O(m\log n)$	$O(\log^2 m)$
	$T_1 = O(m\log^2 n)$	$T_\infty = D\log^3 m$ (D = diameter of graph)

- **Map-reduce model:**
  - separate memories
  - loosely synchronised
  - data exchanged over fast interconnect
  - **Data:** (key, value) pairs on distributed shared disk/Filesystem
  - **Round:** (Metric: how many rounds needed? (best: $O(1)$))
    - map(key, value) → (key, value)
    - shuffle (group items by key)
    - reduce(key, [values...]) → (key, value)
  - can optionally produce any number of pairs, but try not to have input/output of more than $O(n^\varepsilon)$ pairs, or use more than $O(n^\varepsilon)$ time/memory.
  - each (key, value) pair should be $O(\text{polylog}(n))$.
- **Nice properties to have:**
  - **Associative reducer:** scheduler can perform reduce of the same key on multiple threads
  - **Certain (sentinel) pairs come first:** e.g. select from multiple datasets
- E.g. Bellman Ford with n iterations (Stream) (assuming small max degree)