
Advanced Algorithms: Problems and Solutions

Mattia Setzu

Giorgio Vinciguerra

October 2016

Contents

1	Range updates	3
2	Depth of a node in a random search tree	5
2.1	Proof with indicator variable	5
2.2	Recursive balanced proof	6
3	Karp-Rabin fingerprinting on strings	8
3.1	First solution	8
3.2	Second solution	8
4	Hashing sets	11
5	Family of uniform hash functions	13
5.1	First proof	13
5.2	Second proof	13
6	Deterministic data streaming	15
7	Special case of most frequent item in the stream	16
8	Count-min sketch: extension to negative counters	17
8.1	First solution	17
8.2	Second solution	18
9	Count-min sketch: range queries	19
10	Space-efficient perfect hash	21
11	Bloom filters vs. space-efficient perfect hash	23
12	MinHash sketches	24
13	Randomized min-cut algorithm	25
14	External memory implicit searching	26
15	Implicit navigation in vEB layout	27
15.1	Formal description of the rule	28
16	1-D range query	30
A	Hogwarts	31
A.1	Solution 1: Preprocessing-then-Dijkstra	31
A.1.1	Pseudo-code	31
A.2	Solution 2: HogwartsDijkstra	31
A.3	Solution 3: BFS-like traversal	32
B	Paletta	33

1 Range updates

Consider an array C of n integers, initially all equal to zero. We want to support the following operations:

- **update(i, j, c):** where $0 \leq i \leq j \leq n-1$ and c is an integer: it changes C such that $C[k] = C[k] + c$ for every $i \leq k \leq j$.
- **query(i)** where $0 \leq i \leq n-1$: it returns the value of $C[i]$.
- **sum(i, j)** where $0 \leq i \leq j \leq n-1$: it returns $\sum_{k=i}^j (C[k])$.

Design a data structure that uses $O(n)$ space and implements each operation above in $O(\log(n))$ time. Note that $query(i) = sum(i, i)$ but it helps to reason. [Hint to further save space: use an implicit tree such as the Fenwick tree (see wikipedia).]

Solution. Let T be a segmented binary tree over a continuous interval $I : [0, N-1]$ s.t. its leafs are the points in I , and the parent of two nodes comprises of their interval:

$$n' \cup n'' = n, n' \cap n'' = \emptyset \text{ s.t. } n \text{ is the parent of } n', n''$$

T will keep track of the prefix sums for every interval. We define a function

$$s' : [0, n-1] \rightarrow \mathbb{N} \quad (1)$$

that given a node in T returns the value associated with I , namely the cumulative sum of that interval.

In order to reduce the computational cost, we introduce a lazy algorithm that doesn't propagate sums over T as they are streamed in the input, which means $s'(i)$ might not be accurate at a given time t for any of the requested operation.

We'll instead either compute over T or update T as necessary. Let us define a function to do so:

$$l : \mathbb{N} \rightarrow (\mathbb{N} \cup \{\epsilon\}, \mathbb{N}) \quad (2)$$

to keep track of our lazy sums:

$$s(n) = \begin{cases} \epsilon, - & \text{if no lazy prefix sum is in that interval} \\ k, m & \text{if a lazy sum of } k \text{ is to be propagated to } m \end{cases} \quad (3)$$

The QUERY function is then trivial:

```

1: function QUERY( $I, i, sum$ ):
2:   if  $I.size = 1$  then                                     ▷ Return found value
3:     return  $I.sum$ 
4:   if lazy( $I$ ),  $i \in I.left, i \notin I.right$  then             ▷ Lazy on left child
5:     lazy( $I$ )  $\leftarrow$   $False$ 
6:     QUERY( $I.left, i, sum + I.sum$ )
7:   if lazy( $I$ ),  $i \in I.right, i \notin I.left$  then             ▷ Lazy on right child
8:     lazy( $I$ )  $\leftarrow$   $False$ 
9:     QUERY( $I.right, i, sum + I.sum$ )
10:  if lazy( $I$ ),  $i \in I.right, i \in I.left$  then                 ▷ Lazy on both
11:    lazy( $I$ )  $\leftarrow$   $False$ 
12:    QUERY( $I.right, i, j, sum + I.sum$ ) + QUERY( $I.left, i, j, sum + I.sum$ )
13:  if !lazy( $I$ ),  $i \in I.left$  then                               ▷ Not lazy on left child
14:    QUERY( $I.left, i, sum$ )
15:  if !lazy( $I$ ),  $i \in I.right$  then                             ▷ Not lazy on right child
16:    QUERY( $I.right, i, sum$ )
17:  if !lazy( $I$ ),  $i \in I.right, i \in I.left$  then               ▷ Not lazy both
18:    SUM( $I.right, i, sum$ )

```

```

1: function SUM(I, i, j, sum):
2:   if I.size = 1 then                                     ▷ Return
3:     return I.sum + sum
4:   if lazy(I), i ∈ I.left, i ∉ I.right then               ▷ Lazy on left
5:     lazy(I) ← False
6:     SUM(I.left, i, j, sum + I.sum)
7:   if lazy(I), i ∈ I.right, i ∉ I.left then               ▷ Lazy on right
8:     lazy(I) ← False
9:     SUM(I.right, i, j, sum + I.sum)
10:  if lazy(I), i ∈ I.right, i ∈ I.left then               ▷ Lazy on both
11:    lazy(I) ← False
12:    SUM(I.right, i, j, sum + I.sum) + SUM(I.left, i, j, sum + I.sum)
13:  if !lazy(I), i ∈ I.left then                             ▷ Not lazy on both
14:    SUM(I.left, i, sum)
15:  if !lazy(I), i ∈ I.right then                             ▷ Not lazy on both
16:    SUM(I.right, i, sum)
17:  if !lazy(I), i ∈ I.right, i ∈ I.left then             ▷ Not lazy on both
18:    SUM(I.right, i, sum)

1: function UPDATE(I, i, j, k):
2:   if I.size = 1 then                                     ▷ Return
3:     return I.val ← I.val + update
4:     return I.val += update
5:   if lazy(I), i ∈ I.left, i ∉ I.right then               ▷ Lazy on left
6:     lazy(I.left) ← True
7:     I.left.val ← k
8:   if lazy(I), i ∈ I.right, i ∉ I.left then               ▷ Lazy on right
9:     lazy(I.right) ← True
10:    I.right.val ← k
11:  if lazy(I), i ∈ I.right, i ∈ I.left then               ▷ Lazy on both
12:    lazy(I) ← True
13:    I.val ← k
14:  if !lazy(I), i ∈ I.left then                             ▷ Not lazy
15:    UPDATE(I.left, i, update)
16:    update(I.left, i, update)
17:  if !lazy(I), i ∈ I.right then                             ▷ Not lazy
18:    UPDATE(I.right, i, update)
19:    update(I.right, i, update)
20:  if !lazy(I), i ∈ I.right, i ∈ I.left then             ▷ Not lazy
21:    UPDATE(I.right, i, update)
22:    update(I.right, i, update)

```

2 Depth of a node in a random search tree

A random search tree for a set S can be defined as follows: if S is empty, then the null tree is a random search tree; otherwise, choose uniformly at random a key k as root, and the random search trees on $L = \{x \in S : x < k\}$ and $R = \{x \in S : x > k\}$ become, respectively, the left and right subtree of the root k . Consider the randomized QuickSort discussed in class and analyzed with indicator variables CLRS 7.3, and observe that the random selection of the pivots follows the above process, with indicator variables, prove that:

1. the expected depth of a node (i.e. the random variable representing the distance of the node from the root) is nearly $2 \ln n$;
2. the expected size of its subtree is nearly $2 \ln n$ too, observing that it is a simple variation of the previous analysis;
3. the that the probability that the depth of a node exceeds $c 2 \ln n$ is small for any given constant $c > 1$.

2.1 Proof with indicator variable

Prove that the expected depth of a node is nearly $2 \ln n$.

Proof. Let z_m the m th smallest element in S and

$$X_{ij} = \begin{cases} 1 & \text{if } z_j \text{ is an ancestor of } z_i \text{ in the random search tree} \\ 0 & \text{otherwise} \end{cases}$$

The depth of the node i in the tree is given by the number of its ancestors:

$$X = \sum_{\substack{j=1 \\ j \neq i}}^n X_{ij} \tag{4}$$

Note that the depth of a node is also equal to the number of comparison it's involved in (in other words, the number of times it became the left or the right child of a randomly chosen pivot).

Once a pivot k is chosen from S , S is partitioned in two subsets L and R . The elements in the set L will not be compared with the elements in R at any subsequent time. The event $E_1 = z_j$ is an ancestor of z_i in the random search tree occurs if z_j and z_i belongs to the same partition *and* z_j was chosen as pivot before z_i . The probability that E_1 occurs, since it is the intersection of two events, can be upper bounded by:

$$\Pr\{z_j \text{ was chosen as pivot before } z_i\} = \frac{1}{\text{size of the partition}} \leq \frac{1}{|j - i| + 1}$$

because pivots are chosen randomly and independently, and because the partition that contains both z_j and z_i must contain *at least* the $|j - i| + 1$ numbers between z_j and z_i .

Taking expectations of both sides of (4), and then using linearity of expectation, we have:

$$\begin{aligned} \mathbb{E}[X] &= \sum_{\substack{j=1 \\ j \neq i}}^n \mathbb{E}[X_{ij}] \\ &= \sum_{\substack{j=1 \\ j \neq i}}^n \Pr\{z_j \text{ is an ancestor of } z_i \text{ in the random search tree}\} \\ &\leq \sum_{\substack{j=1 \\ j \neq i}}^n \frac{1}{|j - i|} \\ &= \sum_{j=1}^{i-1} \frac{1}{i - j} + \sum_{j=i+1}^n \frac{1}{j - i} \end{aligned}$$

With the change of variables $l = i - j$ and $m = j - i$:

$$= \sum_{l=1}^{i-1} \frac{1}{l} + \sum_{m=1}^n \frac{1}{m} \approx 2 \ln n$$

□

Prove that the expected size of its subtree is nearly $2 \ln n$ too, observing that it is a simple variation of the previous analysis.

Proof. The size of the subtree of a randomly chosen pivot of $z_j \in S$ is given by the number of its descendants. Since (4) is the number of ancestors of a node z_i , we can find the number of descendants of z_j by changing the summation from $j = 1, \dots, n$ to $i = 1, \dots, n$. □

Prove that the probability that the depth of a node exceeds $c \ln n$ is small for any given constant $c > 1$.

Proof. We apply the Theorem 1 below with $(1 + \delta) = c$ and $\mu = 2 \ln n$:

$$\mathbb{P}[X > 2c \ln n] < \left(\frac{e^{c-1}}{c^c} \right)^{2 \ln n}$$

Since $\lim_{n \rightarrow +\infty} a^n = 0$ with $0 < a < 1$, and $\lim_{n \rightarrow +\infty} 2 \ln n = +\infty$, we have to prove that, for any $c > 1$

$$0 < \frac{e^{c-1}}{c^c} < 1$$

The first inequality is simple to verify, for the latter observe that

$$\frac{e^{c-1}}{c^c} < 1 \iff e^{c-1} < c^c \iff c - 1 < c \ln c \iff c(1 - \ln c) < 1.$$

□

Theorem 1 (Chernoff Bound). *Let X_1, \dots, X_n be independent Poisson trials such that, for $1 \leq i \leq n$, $\mathbb{P}[X_i = 1] = p_i$, where $0 < p_i < 1$. Then, for $X = \sum_{i=1}^n X_i$, $\mu = \mathbb{E}[X] = \sum_{i=1}^n p_i$ and any $\delta > 0$,*

$$\mathbb{P}[X > (1 + \delta)\mu] < \left(\frac{e^\delta}{(1 + \delta)^{(1 + \delta)}} \right)^\mu.$$

2.2 Recursive balanced proof

Let n be the number of nodes in the input list l , $h = \log_2(n)$ the height of a balanced tree over l , $T(p)$ the tree built over the permutation p of pivots, $d(m)$ be the positional distance of a value m of a partition from the median value of the said partition. Then the following holds:

- $height(T) = h \iff |T.left| = |T.right| \pm 1$ Trivially, let r be the root of a 3-nodes partition: then, if the partition is unbalanced, the lesser one will comprise of 0 nodes, while the greater one of 2, which implies that $height(T.right) == 2$.
- $P = \text{pivot}, d(m) = \pm k \implies height(T.left) = height(T.right) \pm k$. Recursively from the previous statement, a partition unbalanced of one element generates subtrees whose levels differ on a factor of 1. By iterating recursively, their subtrees, if unbalanced by 1, will yield one more level difference. Over k unbalanced pivots on a single subtree, at most k levels will be added to h .
- By the previous statement, it follows that $\nexists T, T' : height(T) >= height(T'), T \text{ balanced}, T' \text{ unbalanced}$. As stated, let T', T be the unbalanced/balanced tree respectively; let us cheat with T and switch the root pivot with the first element in its subtree. Now, let us prove by contradiction that T can't stay balanced and that its height will increase. By shifting the tree to the left we have deprived $T.right$ of either 0 levels (in case $T.right$ is able to switch every pivot in its tree with its right subtree root, ending with the rightmost leaf in its subtree) or 1, in case no rightmost leaf is present. Therefore $height(T) <= height(T')$.

- The completely unbalanced tree is the tree with the most levels. By taking partitions of size 0 we constantly force, at each level, one subtree to disappear. Therefore, its level(s) has to be necessarily transferred to its brother. We then have exactly one node per level, therefore n levels.

Behaviour on random permutations Now let us analyse how the tree depth varies according to random pivot selection. We start by applying the 2.2k-distance to a tree T with $n = 3$ nodes. Trivially, $height(T)$ with balanced tree is equal to two. Now, let us pick either the lowest or the greatest pivot possible: the tree is unbalanced towards either the left or the right, but $height(T) = 2$ in both cases. As the reader can see from 2.2, the distance works in absolute value; it is then clear how, at every permutation for a pivot p , out of the n , there are 2 that generate a tree of the same height: $p = d(P) + k, p = d(P) - k$. Given that at every iteration a node x in a completely unbalanced tree T' has a probability of $\frac{1}{n-i}$, we can define the probability of x being a pivot at level l as:

$$P(x_k) = \frac{1}{n-l} \quad (5)$$

Now, in order for x not to be chosen as pivot in the previous $l-1$ levels we have:

$$P(x_k) = \sum_{k=1}^{l-1} \left(\frac{1}{n-l+1} \right) \quad (6)$$

Given the height of T , the (harmonic) partial series converges to $\ln(n) + 1$. Let us now add a root r s.t. $T'.right = T, T'.left = T$. We now have to consider the mirror case $\ln(n') + \ln(n')$, given by the previous $n' = n/2$ in the logarithm, since the number of nodes doubled, the $+1$ removed for both, since now neither of $T'.left, T.right$ is the root, and a $+1$ added since a new level has been added.

Upper bound. By hypothesis,

$$\mathbb{E}[d(x) > 2c \ln(n)] \ll 1 \quad (7)$$

By definition the ancestor of a node i are independent random variables, and we can apply the Chernoff bounds over the set $x : d(x) \geq 2 \ln(n)$ of random variables determining the expected distance of nodes.

$$\mathbb{P}[X \geq c \mathbb{E}[X]] < e^{-c \ln(\frac{c}{e}) \mathbb{E}[X]}$$

Let us consider $X = 1 \forall i == \ln(n)$, the expected depth of $\ln(n)$, then

$$\mathbb{P}[X \geq c \ln(n)] < e^{-c \ln(\frac{c}{e}) \ln(n)}$$

3 Karp-Rabin fingerprinting on strings

Given a string $S \equiv S[0 \dots n-1]$, and two positions $0 \leq i < j \leq n-1$, the longest common extension $lce_S(i, j)$ is the length of the maximal run of matching characters from those positions, namely: if $S[i] \neq S[j]$ then $lce_S(i, j) = 0$; otherwise, $lce_S(i, j) = \max\{l \geq 1 : S[i \dots i+l-1] = S[j \dots j+l-1]\}$. For example, if $S = \text{abracadabra}$, then $lce_S(1, 2) = 0$, $lce_S(0, 3) = 1$, and $lce_S(0, 7) = 4$. Given S in advance for preprocessing, build a data structure for S based on the Karp-Rabin fingerprinting, in $O(n \log n)$ time, so that it supports subsequent online queries of the following two types:

- $lce_S(i, j)$: it computes the longest common extension at positions i and j in $O(\log n)$ time.
- $equal_S(i, j, l)$: it checks if $S[i \dots i+l-1] = S[j \dots j+l-1]$ in constant time.

Analyze the cost and the error probability. The space occupied by the data structure can be $O(n \log n)$ but it is possible to use $O(n)$ space. [Note: in this exercise, a one-time preprocessing is performed, and then many online queries are to be answered on the fly]

3.1 First solution

In order to save computational cycles on checks over ranges we use a similar structure to the one in the range updates: we compute the hashing on the first character in $O(1)$ time, then roll the hash through the $n-1$ remaining characters through $nO(1)$ operations. We call H this array; we also denote h_k as the function ca^i computing the Rabin-Karph hash of a string s . The reader shall now see that $\exists h^{-1}(s)$: that is, h is invertible in $O(1)$. The entries $h[i] = \sum_{i \in [0, n-1]} (h(i))$ have cumulative hash and the following properties hold:

- $h[s[i]] = (h[i] - h[i-1])/a^{-1}, a^{-1} = a^1$
- $h[i..j] - h[k..l] = (h[l] - h[k-1])/a^{-1} - (h[j] - h[i-1])/a^{-1}, a^{-1} = \text{modular inverse}$

EQUALS works on cumulative hashes, subtracting them and scaling them accordingly, as our *rabin* function multiplies by an a^i costant.

```

1: function EQUALS( $i, j, length$ ):
2:    $h_i = h[i + length] - h[i - 1]$ 
3:    $h_j = h[j + length] - h[j - 1]$ 
4:    $h^i = h_i / inv(a, i, l)$ 
5:    $h^j = h_j / inv(a, j, l)$ 
   return  $h^i - h^j == 0$ 
6: function INV( $h, k, l$ ): return  $h^{k-l}$ 

```

LCE works on cumulative hashes, checks the equality on the middle element of the strings and runs recursively on the half with different hashing. We define LCE as an auxiliary function

```

1: function LCE( $i, j, l$ ):
2:    $eq = \text{EQUALS}(i, j)$ 
3:   if  $eq$  then return  $l$ 
4:   else if  $\neg \text{EQUALS}((j-i)/2, (n-j)/2, l)$  then return  $\text{EQUALS}((j-i)/2, (n-j)/2)$ 
5:   else  $(j-i)/2 + \text{return EQUALS}((j-i)/2, (n-j)/2)$ 
6:    $h_i = h[i + length] - h[i - 1]$ 
7:    $h_j = h[j + length] - h[j - 1]$ 
8:    $h^i = h_i / inv(a, i, l)$ 
9:    $h^j = h_j / inv(a, j, l)$ 
   return  $h^i - h^j == 0$ 
10: function INV( $h, k, l$ ): return  $h^{k-l}$ 

```

3.2 Second solution

We interpret the string $S \in \Sigma^*$ as the polynomial $S(z) = \sum_{i=0}^{n-1} S[i]z^i$, where $S[i]$ is a symbol encoded with a number in $\{0, \dots, |\Sigma| - 1\}$. Given a prime number p greater than both $2n$ and $|\Sigma|$, we define the

fingerprint on strings h_p as:

$$h_p(S) = h_p \left(\sum_{i=0}^{n-1} S[i] \cdot z^i \right) = \left(\sum_{i=0}^{n-1} S[i] \cdot z^i \right) \bmod p$$

where z is randomly chosen from \mathbb{Z}_p .

If two strings A and B are equal, then also their fingerprints are equal. If they are different, the probability that their fingerprints are the same is at most $\frac{n}{p} < \frac{n}{2n} = \frac{1}{2}$, because there are p choices for z and at most n roots of the polynomial $P(z) = A(z) - B(z)$ (follows from the fundamental theorem of algebra and the fact that $A(z)$ and $B(z)$ have both degree n) [4, p. 169].

Queries. To implement efficiently the queries on S , we use an array H with the hashes of the string's prefixes, formally:

$$H[i] = h_p(S[0..i]) \quad \forall i \in [0, n-1]$$

where $S[i..j]$ denotes the substring $S[i]S[i+1]\dots S[j]$.

With the array H we can compute in constant time the fingerprint of any substring of S , hence we can decide — with a probability of error — $\text{equals}_S(i, j, l)$, by comparing the substrings' fingerprints. To compute $\text{lce}_S(i, j)$, we search with $O(\log n)$ different values of l inside the interval $[0, \min(j-i, n-j)]$ until the result is found. Listing 1 shows an implementation in Python.

Analysis. H is built in $O(n)$ time, because at each iteration the fingerprint in $H[i]$ is computed in constant time from the fingerprint in $H[i-1]$. The space needed by the solution is $O(n \log p)$, because we need to store the array H , that has n entries of size $\log_2 p$.

Listing 1: Source code to compute the longest common extension efficiently.

```

1 p = 179426549
2 z = 3
3 s = "abracadabra"
4
5 def h(s):
6     return sum(ord(c) * z ** i for i, c in enumerate(s)) % p
7
8 H = [ord(s[0]) % p] # Array with hashes of prefixes
9 exp = z
10 for i in range(1, len(s)): # O(n)
11     H.append((H[i-1] + (ord(s[i]) * exp) % p))
12     exp = exp * z
13
14 def rollLeft(j, count): # Compute the hash of the string s[j:j+count]
15     assert count > 0 and j + count <= len(s)
16     if j == 0:
17         return H[count-1]
18     return (H[j+count-1] - H[j-1]) / z ** j
19
20 def equal(i, j, l): # O(1)
21     assert i < j and j + l - 1 < len(s) and l > 0
22     hs1 = rollLeft(i, l) # hash of s[i:i+l]
23     hs2 = rollLeft(j, l) # hash of s[j:j+l]
24     return hs1 == hs2
25
26 def lceaux(i, j, intsize, maxlce): # O(log n)
27     if (intsize <= 1):
28         return 1 if equal(i, j, 1) else 0
29     if equal(i, j, intsize): # Check if there is a longer common extension
30         if (intsize + intsize / 2 >= maxlce):

```

```

31         return maxlce
32     return lceaux(i, j, intsize + intsize / 2, maxlce)
33 else: # Check if there is a shorter common extension
34     return lceaux(i, j, intsize - intsize / 2, intsize)
35
36 def lce(i, j):
37     maxlce = min(j - i, len(s) - j)
38     if (equal(i, j, maxlce)):
39         return maxlce
40     return lceaux(i, j, maxlce, maxlce)

```

4 Hashing sets

Your company has a database $S \subseteq U$ of keys. For this database, it uses a randomly chosen hash function h from a universal family H (as seen in class); it also keeps a bit vector B_S of m entries, initialized to zeroes, which are then set $B_S[h(k)] = 1 \forall k \in S$ (note that collisions may happen). Unfortunately, the database has been lost, thus only B_S and h are known, and the rest is no more accessible. Now, given $k \in U$, how can you establish if k was in S or not? What is the probability of error? (Optional: can you estimate the size $|S|$ of S looking at h and B_S and what is the probability of error?) Later, another database R has been found to be lost: it was using the same hash function h , and the bit vector B_R defined analogously as above. Using h, B_S, B_R , how can you establish if k was in $S \cap R$ (union), $S \cup R$ (intersection), or $S \setminus R$ (difference)? What is the probability of error?

Solution. Trivially for $B_S[h(k)] = 0$ we can answer FALSE with $\mathbb{P}[\text{error}] = 0$. Let us analyse the opposite case, $B_S[h(k)] = 1$. Let $i \in [0, m]$ be some index s.t. $B_S[i] = 1$, and let $cl_S(i)$ be the list of $k \in S : h(k) = i$ for some set $S \in \mathcal{P}(S)$. We can then denote the sets $cl_U := \{k_U : k_U \in U, h(k_U) = i\}$, $cl_S := \{k_S : k_S \in S, h(k_S) = i\}$; it is trivial to show that

- $cl_S \subseteq cl_U$ as $S \subseteq U$.
- $|cl_S(k)| \leq |cl_U(k)| \forall k \in U$ as $S \subseteq U$.

Let us not try and estimate $|cl_S(k)|$: given h is universal, we have an expected value of collisions of $\mathbb{E}[X_k] \approx \frac{1}{m} \forall k \in S$, that is

$$\begin{aligned} \mathbb{P}[h(k^0) = c] &= \frac{1}{m} \\ \mathbb{P}[h(k^1) = c] &= \frac{1}{m^2} \\ \mathbb{P}\left[h(k^{i-1}) = c \mid h(k^i) = c\right] &= \frac{1}{m^i} \\ \mathbb{P}[h(k) = c, \forall k \in S] &= \frac{1}{m^{|S|}} \end{aligned}$$

We can similarly compute the probability of not collision by simply replacing $\frac{1}{m}$ with $(1 - \frac{1}{m})$: $\mathbb{P}[h(k) \neq c, \forall k \in S] = 1 - \frac{1}{m^{|S|}}$.

Given our estimate of the collision list, we can now compute an estimate of the error probability: by 4, we give an erroneous answer whenever $k \in cl_U(k), k \notin cl_S(k)$, that is we have a margin of error of $cl_U(k) \setminus cl_S(k)$ whose size is $|cl_U(k)| - |cl_S(k)|$. Given the set of k for which $h(k) = i$ the *bad answers* are then

$$1 - \mathbb{P}[\text{good answer}] = 1 - \left(1 - \frac{1}{m}\right)^{|S|} \quad (8)$$

We now provide a lower and upper bound for the said value. The lower bound is given by the *perfect hash* with no collisions: given $e =$ number of 1 in B_S , e is the lowerbound. Provided that $m \geq c|S|$ for some $c > 1$, $|S| \leq \frac{m}{c}$:

$$\mathbb{P}[\text{collision over } i] = \alpha_S = \frac{\frac{m}{c}}{m} = \frac{1}{c} \quad (9)$$

Therefore

$$e \leq 1 - \frac{1}{m^{|S|}} \leq \frac{1}{c} \quad (10)$$

Expected number of 1 in B_S . We define a random variable X_k :

$$X_k = \begin{cases} 1 & \text{if } B_S[k] = 1 \\ 0 & \text{otherwise} \end{cases}$$

and one over the ones in B_s , X :

$$X = \sum_{k=0}^{m-1} X_k \tag{11}$$

We compute the relative expected value $\mathbb{E}[X]$:

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E}\left[\sum_{k=0}^{m-1} X_k\right] \\ &= \sum_{k=0}^{m-1} \mathbb{P}[B_s[k] = 1] \\ &= \sum_{k=0}^{m-1} \frac{\alpha}{m} \leq \frac{m}{c} \end{aligned}$$

where $\mathbb{P}[B_s[k] = 1] = \frac{\alpha}{m}$ as α are the favorable cases (i.e. the collisions for a generic bucket $B_s[i]$), and m is the size of B_s .

5 Family of uniform hash functions

The notion of pairwise independence says that, for any $x_1 \neq x_2, c_1, c_2 \in \mathbb{Z}_p$, we have that

$$\mathbb{P}[h(x_1) = c_1, h(x_2) = c_2] = \mathbb{P}[h(x_1) = c_1] \times \mathbb{P}[h(x_2) = c_2] \quad (12)$$

In other words, the joint probability is the product of the two individual probabilities. Show that the family of hash functions $\mathcal{H} = \{h_{a,b}(x) = ((ax+b) \bmod p) \bmod m : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$ is *pairwise independent* where p is a sufficiently large prime number ($m+1 \leq p \leq 2m$).

5.1 First proof

By linear algebra, $a + (kp) \bmod p = c \forall k \in \mathbb{N}$. If we were to cap $a + pk \bmod p = c \forall k \in K, K_N = k_i : k_i < N$

$$\mathbb{P}[h(x_i) = c_i] = \frac{1}{m^2} = \mathbb{P}[h(x_1) = c_1, h(x_2) = c_2]$$

Given x_i, d , we define as $m_i = (ax_i + b)$; since $(ax_i + b)$ is a *linear transformation* m_i is unique. It follows trivially that there are $\frac{p}{m}$ values for which $m_i \bmod p = d$ since $p > m$ and by 5.1. The same goes for x_1, x_2 :

$$\begin{aligned} (ax_1 + b) \bmod p &= d \\ (ax_2 + b) \bmod p &= e \end{aligned}$$

By the Chinese remainder theorem the above system has only one solution for the variable (a, b) over the $(p)(p-1)$ possible pairs, therefore $(ax_1 + b) \bmod p = d$ and $(ax_2 + b) \bmod p = e$ for $\approx \frac{p}{m}$ cases each. Since d, e are independent

$$((ax_2 + b) \bmod p = d) \wedge ((ax_2 + b) \bmod p = e) \text{ for } \frac{p}{m}, \frac{p}{m} = \frac{p^2}{m^2}$$

Over all the possible choices of (a, b) :

$$\mathbb{P}[(ax_2 + b) \bmod p = d \wedge (ax_2 + b) \bmod p = e] = \frac{\frac{p^2}{m^2}}{p(p-1)}$$

We now prove $\mathbb{P}[h(x_1) = c_1] = \frac{1}{m}$. Of all the possible m buckets, one and only one is the one we look for: $\mathbb{P}[l \bmod m = c_1] = \frac{1}{m}$. As we've seen by 5.1 at most $\frac{p}{m}$ such l exist for $(ax + b) \bmod p$, therefore $\mathbb{P}[h(x_i) = c_i] = \frac{\frac{p}{m}}{m} \approx \frac{1}{m}$. Which computes $\approx \frac{1}{m^2}$ and proves the assumption under the said approximation.

5.2 Second proof

Proof. We will show that both sides of (12) are approximately equal to $\frac{1}{m^2}$.

Given a hash function $h_{a,b} \in \mathcal{H}$, and two distinct inputs $x_1, x_2 \in \mathbb{N}$, let:

$$\begin{aligned} r &= (ax_1 + b) \bmod p \\ s &= (ax_2 + b) \bmod p \end{aligned}$$

Notice that $r \neq s$ because $r - s \equiv a(x_1 - x_2) \pmod{p}$, both a and $x_1 - x_2$ are nonzero modulo a prime number, and so their product is nonzero.

Since there are a total of $p(p-1)$ pairs of (r, s) such that $r \neq s$ (p^2 choices subtracted the number of pairs where $r = s$), there is a one-to-one correspondence between pairs $(a, b) \in \mathbb{Z}_p^* \times \mathbb{Z}_p$ and pairs (r, s) . Therefore, for the left-hand side of (12):

$$\mathbb{P}_{h_{a,b} \in \mathcal{H}}[(h(x_1) = c_1, h(x_2) = c_2)] = \mathbb{P}_{r \neq s \in \mathbb{Z}_p^2}[r \bmod m = c_1, s \bmod m = c_2]$$

There are about p/m values of r that satisfy $r \bmod m = c_1$, and the same goes for c_2 and s . Hence, there are about $(p/m)^2$ choices for the pair $(r, s) \in \mathbb{Z}_p^2$ that satisfy $r \bmod m = c_1 \wedge s \bmod m = c_2$. Dividing the favorable cases with the all possible cases:

$$\frac{\lfloor p/m \rfloor^2}{p(p-1)} \leq \mathbb{P}[r \bmod m = c_1, s \bmod m = c_2] \leq \frac{(\lfloor p/m \rfloor + 1)^2}{p(p-1)}$$

Thus, $\mathbb{P}[h(x_1) = c_1, h(x_2) = c_2] \approx \frac{1}{m^2}$.

The right-hand side of (12) is equal to $\frac{1}{m^2}$, because both probabilities are $\frac{1}{m}$. Take for example $\mathbb{P}[h(x_2) = c_2]$: there are about p/m values of s that satisfy $s \bmod m = c_2$, out of p values for s . Hence:

$$\mathbb{P}[h(x_2) = c_2] = \mathbb{P}[s \bmod m = c_2] \approx \frac{\frac{p}{m}}{p} = \frac{1}{m}$$

□

6 Deterministic data streaming

Consider a stream of n items, where items can appear more than once. The problem is to find the most frequently appearing item in the stream (where ties are broken arbitrarily if more than one item satisfies the latter). For any fixed integer $k \geq 1$, suppose that only k items and counters can be stored, one item per memory cell, where each counter can use only $O(\text{polylog}(n))$ bits (i.e. $O(\log^c n)$ for any fixed constant $c > 0$): in other words, only $b = O(k \cdot \text{polylog}(n))$ bits of space are available. (Note that, even though we call them counters, they can actually contain any kind of information as long as it does not exceed that amount of bits.) Show that the problem cannot be solved deterministically under the following rules: the algorithm can only use b bits, and read the next item of the stream, one item at a time. You, the adversary, have access to all the stream, and the content of the b bits stored by the algorithm: you cannot change those b bits and the past, namely, the items already read by the algorithm, but you can change the future, namely, the next item to be read. Since the algorithm must be correct for any input, you can use any amount of streams to be fed to the algorithm and as many distinct items as you want. [Hint: it is an adversarial argument based on the fact that, for many streams, there can be a tie on the items.]

Solution. We, the adversaries, can decide the alphabet Σ and the content of the stream $s \in \Sigma^*$. The *idea* is to create a stream generator that forces any deterministic algorithm to reach a state q with at least two different streams. In fact, the number of states (i.e. configurations of memory) of this class of algorithms is bounded by b and, by the pigeonhole principle, there exist at least two streams that cause the algorithm to transition the same state.

Suppose that q is reached by the three streams:

1. $\alpha 0011$, where $\alpha \in \Sigma^*$ contains the same number of 0s and 1s
2. $\beta aabb$, where $\beta \in \Sigma^*$ contains the same number of a s and b s
3. $\gamma 00bb$, where $\gamma \in \Sigma^*$ contains the same number of 0s and b s

When we will emit 0 as next character of the stream and query the algorithm for the most frequent item, we will receive an answer \mathcal{A} . The algorithm is deterministic, therefore in q the answer will be the same for each stream. But \mathcal{A} will be wrong for at least one of the three streams. For example, if $\mathcal{A} = 0$, then the answer is wrong for the second stream.

7 Special case of most frequent item in the stream

Suppose to have a stream of n items, so that one of them occurs $> \frac{n}{2}$ times in the stream. Also, the main memory is limited to keeping just two items and their counters, plus the knowledge of the value of n beforehand. Show how to find deterministically the most frequent item in this scenario.

Hint: since the problem cannot be solved deterministically if the most frequent item occurs $\leq \frac{n}{2}$ times, the fact that the frequency is $> \frac{n}{2}$ should be exploited.

Solution. We'll use the notation of the previous section. Before illustrating our solution we prove that the following hold:

- **Given the j^1 element, $C(j^1) > C(j^i) \forall i \in S$.**
- Following the previous, $C(j^i) < C(j^i), i > 1$ element.
- **There are at most $\frac{n}{2} - 1$ elements in S :** given that j^1 appears at least $\frac{n}{2} + 1$ times, in the best case scenario every other $\frac{n}{2} - 1$ element is different, giving $\frac{n}{2} - 1$ different elements.
- **j^1 has sub-stream dominance:** $\exists S' \subseteq S : j^1$ is the most frequent item $\in S'$: trivially, since it is the most frequent element, S' is always a sub-stream dominant sub-stream.

Recursive sub-streams It is trivial to show that if we were to have $\frac{n}{2}$ counters, $\sum_{i=0, i \neq 1}^{\frac{n}{2}} (C(j^i) < C(j^1))$ by 7. Though trivial, by combining it with 7 we can assert that given $S' = S[0, \frac{n}{2}]$, $S'' = S[\frac{n}{2}, n]$, $k =$ number of elements $\in S$

- j^1 is the sub-stream dominant element in either S', S'' : by contradiction, let us assume that is false. Then $\exists j_o \neq j^1 : C(j_o) \text{ in } S' > C(j^1), \exists j_o : C(j_o) \text{ in } S'' > C(j^1)$; given that $S'S'' = S$, that would make $C(j_o) = C(j^1)$: contradiction.
- More generally, given two sub-streams S', S'' , the sub-stream dominant element of $S'S''$ is one of the sub-stream dominants in S' or S'' and its frequencies are defined by

$$e - k - 1 \text{ in } S', \frac{n}{2} + 1 - (e - k - 1) \text{ in } S''$$

where e is an arbitrary number of appearances of $j^2 : e < \frac{n}{2} + 1$ The reader will see as the $+1$ in the right side guarantees the sub-stream dominant to be maximum. The sides can be swapped without hurting generality.

In order to better illustrate, we provide an algorithm (the Boyer-Moore majority vote algorithm) that exploits the sub-stream dominance. Our idea is to *go up* with our counter when we are fed an object we've already met, and to *go down* when we are fed an object different from us. Once we reach the bottom (0), we know by 7 that we either reached zero for an object $\neq j^1$, and we can ignore it, or we reached zero for j^1 , that being the most frequent element will have sub-stream dominance in the remaining sequence, which means it will *go upper* than any other element in that sub-stream. Given that any other element has lower frequency, the most frequent of them, j^2 will *go down* $e < \frac{n}{2} + 1$ times.

```

1: function MAJORITY_VOTE_ALGORITHM( $S$ ):
2:    $candidate \leftarrow \text{null}$ 
3:    $counter \leftarrow 0$ 
4:   for all  $object \in S$  do
5:     if  $counter = 0$  then
6:        $candidate \leftarrow object$                                  $\triangleright$   $candidate$  is no more dominant, swap it
7:     else if  $object = candidate$  then
8:        $counter \leftarrow counter + 1$ 
9:     else if  $object \neq candidate$  then
10:       $counter \leftarrow counter - 1$ 
11:  return  $candidate$ 

```


8 Count-min sketch: extension to negative counters

Check the analysis seen in class, and discuss how to allow $F[i]$ to change by arbitrary values read in the stream. Namely, the stream is a sequence of pairs of elements, where the first element indicates the item i whose counter is to be changed, and the second element is the amount v of that change (v can vary in each pair). In this way, the operation on the counter becomes $F[i] = F[i] + v$, where the increment and decrement can be now seen as $(i, 1)$ and $(i, -1)$.

8.1 First solution

Let F be the implicit vector of size n that receives updates (i, v) such that $F[i] = F[i] + v$ (initially, $F[i] = 0 \forall i \in [1, n]$). The Count-Min (CM) sketch data structure, with parameters ε and δ , consists of:

- a table T of size $r \times c$, where $r = \log \frac{1}{\delta}$ and $c = \frac{\varepsilon}{\varepsilon}$, that we use to approximate queries on F .
- r hash functions h_1, \dots, h_r , chosen uniformly at random from a pairwise-independent family, that maps from $\{1, \dots, n\}$ to $\{1, \dots, c\}$.

The parameters specify that the error in answering a query is in within a factor of ε with probability $1 - \delta$. When an update (i, v) arrives, the table is updated as follows:

$$T[i, h_j(i)] = T[i, h_j(i)] + v \quad \forall j = 1, \dots, r$$

A point query returns an approximation $\tilde{F}[i]$ of $F[i]$. If we assume that $F[i] \geq 0$, then:

$$\tilde{F}[i] = \min_{j \in [1, r]} T[j, h_j(i)]$$

with the guarantees:

1. $\tilde{F}[i] \geq F[i]$
2. $\mathbb{P}[\tilde{F}[i] > F[i] + \varepsilon \|F\|] \leq \delta$.

The problem statement asks to remove the assumption $F[i] \geq 0$. In this case the point query becomes:

$$\tilde{F}[i] = \text{median}_{j \in [1, r]} T[j, h_j(i)]$$

with the guarantee that $\mathbb{P}[F[i] - 3\varepsilon \|F\| \leq \tilde{F}[i] \leq F[i] + 3\varepsilon \|F\|] < 1 - \delta^{1/4}$.

Proof. Let $I_{i,j,k}$ the indicator variable that is 1 if $i \neq k \wedge h_j(i) = h_j(k)$ and 0 otherwise. Observe that, by pairwise independence of the hash functions,

$$\mathbb{E}[I_{i,j,k}] = \mathbb{P}[h_j(i) = h_j(k)] \leq \frac{1}{c} = \frac{\varepsilon}{e}$$

and, given $X_{i,j} = \sum_{k=1}^n I_{i,j,k} F[k]$

$$\mathbb{E}[|X_{i,j}|] = \mathbb{E}\left[\sum_{k=1}^n I_{i,j,k} |F[k]| \right] \leq \sum_{k=1}^n |F[k]| \mathbb{E}[I_{i,j,k}] \leq \frac{\varepsilon}{e} \|F\| \quad (13)$$

The probability that any count is off by more than $3\varepsilon \|F\|$ is:

$$\begin{aligned} & \mathbb{P}[|T[i, h_j(i)] - F[i]| \geq 3\varepsilon \|F\|] \\ &= \mathbb{P}[|F[i] + X_{i,j} - F[i]| \geq 3\varepsilon \|F\|] && \text{by construction of } T \\ &= \mathbb{P}[|X_{i,j}| \geq 3\varepsilon \|F\|] \\ &\leq \frac{\mathbb{E}[|X_{i,j}|]}{3\varepsilon \|F\|} && \text{by the Markov inequality} \\ &\leq \frac{\varepsilon \|F\|}{e} \frac{1}{3\varepsilon \|F\|} && \text{by (13)} \\ &= \frac{1}{3e} < \frac{1}{8} \end{aligned}$$

Applying Chernoff bounds...

□

8.2 Second solution

We trivially have some changes over X_{ji} and $\tilde{F}[i]$:

$$\begin{aligned} X_{ji} &= \Sigma^n(I_k F[k]) \\ \tilde{F}[i] &= F[i] + X_{ji} \end{aligned}$$

X_{ji} can vary as complementary increments $(+v_i, +v_j, +v_k, -v_k, -v_j, -v_i)$ can make it ≤ 0 without necessarily being updates on i . In order to have a minimum error estimate we'll pick the *median* value of X_{ji} .

$\tilde{F}[i] = F[i] + X_{ji}$ by the above assertion the counter $\tilde{F}[i]$ could have no garbage, or negative garbage according to the other increments. Therefore by choosing the minimum $\tilde{F}[i]$ over the $\log(\frac{1}{\delta})$ we could actually pick the most perturbed result (think of a collision with the biggest negative increment over one row).

The last step is the probability proof: $\mathbb{P}[\tilde{F}[i] > F[i] + \epsilon\|F\|]$. Since by 8.2 X_{ji} can be either positive or negative we pick the *median* value *med* over the minimum m and maximum M . We are then able to split our error analysis in two of equivalent size $\frac{r}{2}$:

$$\begin{aligned} \mathbb{P}[\tilde{F}[i] \in F[i] + \epsilon\|F\|] &= \\ \mathbb{P}[\tilde{F}[i] \geq F[i] - |\epsilon\|F\| \wedge \tilde{F}[i] \leq F[i] + |\epsilon\|F\|] &= \\ \mathbb{P}[\tilde{F}[i] \geq F[i] - |\epsilon\|F\|) \mathbb{P}[(\tilde{F}[i] \leq F[i] + |\epsilon\|F\|)] & \end{aligned}$$

Let us define $p_0 = \mathbb{P}[\tilde{F}[i] \geq F[i] - |\epsilon\|F\|]$ and $p_1 = \mathbb{P}[\tilde{F}[i] \leq F[i] + |\epsilon\|F\|]$. We can report the analysis seen in class adjusted for the number of rows $\frac{r}{2}$

$$\begin{aligned} p_0 &= \prod^{\frac{r}{2}} \mathbb{P}[|X_{ji}| \geq -|\epsilon\|F\|] = \\ &= -\frac{1}{2^{\frac{r}{2}}} = -\delta^{\frac{1}{2}} \end{aligned}$$

Now for p_1 :

$$\begin{aligned} p_1 &= \mathbb{P}[\tilde{F}[i] \leq F[i] + |\epsilon\|F\|] = \\ 1 - \mathbb{P}[\tilde{F}[i] \geq F[i] + |\epsilon\|F\|] &= 1 - \frac{1}{2^{\frac{r}{2}}} = 1 - \delta^{\frac{1}{2}} \end{aligned}$$

Giving us a combined probability of $p = p_0 p_1 = (-\delta^{\frac{1}{2}})(1 - \delta^{\frac{1}{2}}) = -\sqrt{\delta} + \delta$.

9 Count-min sketch: range queries

Show and analyze the application of count-min sketch to range queries (i, j) for computing $\sum_{k=i}^j F[k]$. Hint: reduce the latter query to the estimate of just $t \leq 2 \log n$ counters c_1, c_2, \dots, c_t . Note that in order to obtain a probability at most δ of error (i.e. that $\sum_{l=1}^t c_l > \sum_{k=i}^j F[k] + 2\epsilon \log n \|F\|$), it does not suffice to say that it is at most δ the probability of error of each counter c_l : while each counter is still the actual wanted value plus the residual as before, it is better to consider the sum V of these t wanted values and the sum X of these residuals, and apply Markov's inequality to V and X rather than on the individual counters.

Solution. A range $[a, b]$ is a dyadic range if its length is a power of two ($l = 2^y$), and begins at a multiple of its own length: $[j2^y + 1, (j+1)2^y]$. For example, $[13, 16]$ can be written as $[3 \cdot 2^2 + 1, (3+1) \cdot 2^2]$. Any arbitrary range of size s can be partitioned into $O(\log s)$ dyadic ranges, for example [2]:

$$[18, 38] = [18, 18] \cup [19, 20] \cup [21, 24] \cup [25, 32] \cup [33, 36] \cup [37, 38]$$

Let n be the size of the implicit vector F whose entries we want to approximate. The idea is to maintain a collection C of $\log_2 n$ CM sketches, one for each set of dyadic ranges of length $2^y \forall y \in [0, \log_2 n - 1]$. The operations on C becomes:

- **Update** ($F[i] = F[i] + v$). Every sketch in C is updated, since each point $1 \leq i \leq n$ is member of $\log_2 n$ dyadic ranges.
- **Range queries** ($\sum_{i=l}^r F[i]$). The range $[l, r]$ is partitioned into at most $2 \log_2 n$ dyadic ranges. For each partition, a point query is made to the corresponding sketch in C ; the (estimated) result of the range query is the sum of the point queries. See Figure 1.

The time to compute the estimate or to make an update is $O(\log n \log \frac{1}{\delta})$. The space used is $O(\frac{1}{\epsilon} \log n \log \frac{1}{\delta})$, because each sketch requires $O(\frac{\epsilon}{\delta} \ln \frac{1}{\delta})$ space [3].

Let $F[l..r] = \sum_{i=l}^r F[i]$ be the answer to the range query and $\tilde{F}[l..r]$ the estimate. The guarantees are:

- $\tilde{F}[l..r] \geq F[l..r]$
- $\mathbb{P} \left[\tilde{F}[l..r] > F[l..r] + 2\epsilon \log n \|F\| \right] \leq \delta$

Proof. Let $I_{i,j,k}$ the indicator variable that is 1 if $i \neq k \wedge h_j(i) = h_j(k)$ and 0 otherwise. From the analysis of CM sketch, we know that $\mathbb{E}[I_{i,j,k}] \leq \frac{\epsilon}{e}$ and

$$\mathbb{E}[X_{i,j}] = \mathbb{E} \left[\sum_{k=1}^n I_{i,j,k} F[k] \right] \leq \frac{\epsilon}{e} \|F\|$$

A range query $\tilde{F}[l..r]$ is assembled by $t \leq 2 \log n$ point queries on dyadic intervals $[l_1, r_1], [l_2, r_2], \dots, [l_t, r_t]$. The expectation of the additive error X^j for any estimator (i.e. the j -th row of any of the t CM sketches queried) is

$$\mathbb{E}[X^j] = \mathbb{E} \left[\sum_{s=1}^t X_{i,j} \right] = \sum_{s=1}^t \mathbb{E}[X_{i,j}] \leq (2 \log n) \frac{\epsilon}{e} \|F\| \quad (14)$$

Also observe that the j -th estimation of $F[l..r]$ is

$$\tilde{F}^j[l..r] = \sum_{s=1}^t \tilde{F}^j[l_s..r_s] = \sum_{s=1}^t (F[l_s..r_s] + X_{i,j}) = F[l..r] + X^j \quad (15)$$

Now, for the j -th estimation we can compute

$$\begin{aligned}
& \mathbb{P} \left[\tilde{F}^j[l..r] > F[l..r] + 2\varepsilon \log n \|F\| \right] \\
&= \mathbb{P} \left[X^j > 2\varepsilon \log n \|F\| \right] && \text{because of (15)} \\
&\leq \frac{\mathbb{E} [X^j]}{2\varepsilon \log n \|F\|} && \text{by the Markov inequality} \\
&\leq \frac{2\varepsilon \log n \|F\|}{e} \frac{1}{2\varepsilon \log n \|F\|} && \text{because of (14)} \\
&= e^{-1}
\end{aligned}$$

Since we have $\ln \frac{1}{\delta}$ independent hash functions $\prod_{j=1}^{\ln(1/\delta)} e^{-1} = e^{-\ln(1/\delta)} = \delta$ □

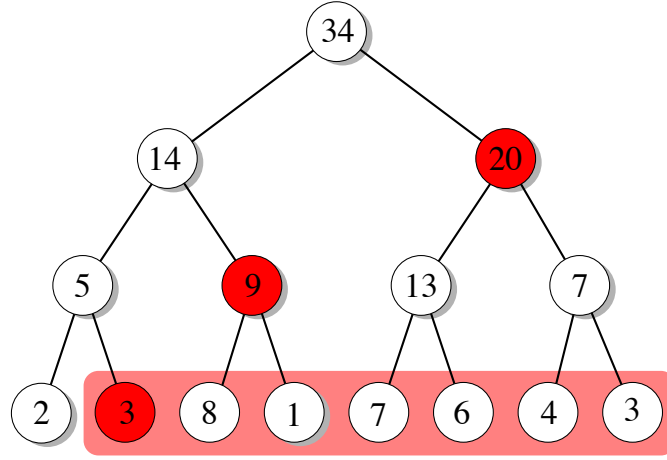


Figure 1: A hierarchy of dyadic ranges. The leaves are the ranges of length 2^0 , while the root corresponds to the single range of length 2^3 . Each level of the tree can be seen as a CM sketch table. To estimate $\sum_{k=2}^8 F[k]$, the range $[2, 8]$ is decomposed into dyadic ranges $[2, 2], [3, 4], [5, 8]$. Each node contains the sum of the values stored in its children. Red nodes are queried and their sum is returned. Adapted from [2].

10 Space-efficient perfect hash

Consider the two-level perfect hash tables presented in [CLRS] and discussed in class. As already discussed, for a given set of n keys from the universe U , a random universal hash function $h : U \rightarrow [m]$ is employed where $m = n$, thus creating n buckets of size $n_j \geq 0$, where $\sum_{j=0}^{n-1} n_j = n$. Each bucket j uses a random universal hash function $h_j : U \rightarrow [m_j]$ with $m_j = n_j^2$. Key x is thus stored in position $h_j(x)$ of the table for bucket j , where $j = h(x)$.

This problem asks to replace each such table by a bitvector of length n_j^2 , initialized to all 0s, where key x is discarded and, in its place, a bit 1 is set in position $h_j(x)$ (a similar thing was proposed in Problem 4 and thus we can have a one-side error). Design a space-efficient implementation of this variation of perfect hash, using a couple of tips. First, it can be convenient to represent the value of the table size in unary (i.e., x zeroes followed by one for size x , so 000001 represents $x = 5$ and 1 represents $x = 0$). Second, it can be useful to employ a rank-select data structure that, given any bit vector B of b bits, uses additional $o(b)$ bits to support in $O(1)$ time the following operations on B :

- $rank_1(i)$: return the number of 1s appearing in the first i bits of B .
- $select_1(j)$: return the position i of the j th 1, if any, appearing in B (i.e. $B[i] = 1$ and $rank_1(i) = j$).

Operations $rank_0(i)$ and $select_0(j)$ can be defined in the same way as above. Also, note that $o(b)$ stands for any asymptotic cost that is smaller than $\Theta(b)$ for $b \rightarrow \infty$.

Solution. The two-level hashing layers are comprised of:

1. A hash function h and n pointers, each addressing one bitvector B_j of size n_j^2 .
2. n hash functions h_j hashing to their relative bitvector B_j of size n_j^2 .

Instead of storing a vector of pointers to the secondary hash tables, we merge the bitvectors B_j , one after the other, in a flat array $B = B_0 B_1 \cdots B_{n-1}$. Note that, except for the n pointers removed at the first level, the space is neither reduced nor increased, as no further bits are used to separate two adjacent buckets. We then store the size of each B_j in unary in an auxiliary bitvector L :

$$L = \underbrace{00 \dots 0 1}_{n_0^2 \text{ times}} \underbrace{00 \dots 0 1}_{n_1^2 \text{ times}} \cdots \underbrace{00 \dots 0 1}_{n_{n-1}^2 \text{ times}}$$

L is associated with a rank-select data structure.

Queries. The starting index of B_j in B is computed in $O(1)$ with the following function on L :

$$\phi(j) = rank_0(select_1(j))$$

This operation:

1. finds the position of the j th 1 with $select_1(j)$, that is, the index in L that precedes the start of the unary representation of n_j^2 ;
2. calculates the sum of the sizes of the preceding bitvectors ($\sum_{i=0}^{j-1} n_i^2$) by computing the number of 0s with $rank_0(select_1(j))$ (this is the starting position of the desired bitvector).

We can determine (with a probability of error) whether a key k belongs to the set $S \subset U$, by testing whether $B[i + h_{h(k)}(k)]$ is equal to 1, where $i = \phi(h(k))$ is the starting position of the $h(k)$ th bitvector, and $h_{h(k)}(k)$ is the offset for the secondary level.

Hash functions space optimization. We now try to improve the space for the hash functions parameters a_j, b_j, p_j . First, we select the lowest p_j possible, that is, the first $p_j > n_j^2$: by Bertrand postulate such a prime p_j exists for $n_j^2 \leq p_j \leq 2n_j^2 - 2$. We therefore need a space of at most $2n_j^2$ bits for each parameter in an unary encoding. Since we have $a \in \mathbb{Z}_{p_j}^*, b \in \mathbb{Z}_{p_j}$, we have a space of $3 \cdot 2n_j^2$ for a triple (a_j, b_j, p_j) of any given hash function. Let us now compute the space for the n hash functions:

$$Y = 3n + 3 \sum_{j=0}^{n-1} 2n_j^2 = 3n + 6 \sum_{j=0}^{n-1} n_j^2$$

where $3n$ is the number of 1s that separate the unary encoding of the parameters. We have also an auxiliary space $o(Y)$ due to the rank-select data structure.

Space. The space occupied by the whole data structure is:

- $X = \sum_{j=0}^{n-1} n_j^2$ bits for B ;
- $X + n$ bits for L , since X is the number of 0s and n is the number of 1s in L ;
- $o(X + n)$ bits for the rank-select auxiliary data structure for L ;
- $Y + o(Y)$ bits for the hash functions, as shown in the previous paragraph.

Since $\mathbb{E} \left[\sum_{j=0}^{n-1} n_j^2 \right] < 2n$, as shown in the perfect hashing analysis [1, p. 281], the total space is $O(n)$.

11 Bloom filters vs. space-efficient perfect hash

Recall that classic Bloom filters use roughly $1.44 \log_2(1/f)$ bits per key, as seen in class (where $f = (1-p)^k$ is the failure probability minimized for $p \approx e^{-\frac{kn}{m}} = 1/2$). The problem asks to extend the implementation required in Problem 10 by employing an additional random universal hash function $s : U \rightarrow [m]$ with $m = \lceil 1/f \rceil$, called signature, so that $s(x)$ is also stored (in place of x , which is discarded). The resulting space-efficient perfect hash table T has now a one-side error with failure probability of roughly f , as in Bloom filters: say why. Design a space-efficient efficient implementation of T , and compare the number of bits per key required by T with that required by Bloom filters.

Solution. We extend the data structure discussed in Section 10 with an additional array C that stores the n signatures. Leveraging the fact that B has as many 1s as the number of signatures we need to store, we fill the array C with the signatures in this way: the entry $C[i]$ contains the signature $s(k)$ in binary if and only if k is the key that caused a the i th one in B . We also need an additional space of $o(X)$ to perform rank-select operations on B in $O(1)$.

With this new data structure, given a key $l \in U$, to check whether it belongs to S we check whether $B[i] = 1$ where $i = \phi(h(l)) + h_{h(l)}(l)$ as before, but also:

$$C[\text{rank}_1(i) - 1] \stackrel{?}{=} s(l). \quad (16)$$

In fact, $\text{rank}_1(i) - 1$, executed on B , returns the position in C where we find the signature for the key $k \in S$ that caused $B[i] = 1$.

We have an error whenever $l \notin S$, but (16) is true. Since the signature has length $1/f$, $\Pr(\text{error}) \leq \frac{1}{1/f} = f$.

Bits per key comparison. If we consider only the space used by C , B and L , our solutions uses $n \log_2 \frac{1}{f} + X + (X + n)$ bits to store n keys. Since $\mathbb{E}[X] < 2n$, on average we need approximately $\log_2 \frac{1}{f} + 5$ bits per key, that compared to the Bloom filters is better for $f < 2^{-\frac{5}{0.44}}$.

12 MinHash sketches

As discussed in class, for a min-wise independent family \mathcal{H} , we can associate a sketch

$$s(X) = \langle \min h_1(X), \min h_2(X), \dots, \min h_k(X) \rangle$$

with each set X in the given data collection, where h_1, h_2, \dots, h_k are independently chosen at random from \mathcal{H} . Consider now any two sets A and B , with their sketches $s(A)$ and $s(B)$. Can you compute a sketch for $A \cup B$ using just $s(A)$ and $s(B)$ in $O(k)$ time? Can you prove that it is equivalent to compute $s(A \cup B)$ from scratch directly from $A \cup B$?

Solution. We claim that

$$\langle \min(\min h_1(A), \min h_1(B)), \dots, \min(\min h_k(A), \min h_k(B)) \rangle$$

which can be computed from $s(A)$ and $s(B)$ with $\Theta(k)$ comparisons, is equivalent to

$$s(A \cup B) = \langle \min h_1(A \cup B), \dots, \min h_k(A \cup B) \rangle$$

In fact, note that $\forall i \in [1, k]$,

$$\min h_i(A \cup B) = \min(h_i(A) \cup h_i(B)) = \min(\min h_i(A), \min h_i(B))$$

The second equality is a trivial property of the union, for $h_i(A \cup B) = h_i(A) \cup h_i(B)$ we give the following

Proof. First we prove that $h_i(A \cup B) \subseteq h_i(A) \cup h_i(B)$:

$$l \in h_i(A \cup B) \implies \exists s \in A \cup B \text{ such that } h_i(s) = l$$

Since $s \in A \cup B \implies s \in A \vee s \in B$, we have two cases: if $s \in A$, then $h_i(s) \in h_i(A)$, hence $h_i(s) \in h_i(A) \cup h_i(B)$; if instead $s \in B$, then $h_i(s) \in h_i(B)$, hence $h_i(s) \in h_i(A) \cup h_i(B)$.

Now we prove $h_i(A) \cup h_i(B) \subseteq h_i(A \cup B)$:

$$l \in h_i(A) \cup h_i(B) \implies l \in h_i(A) \vee l \in h_i(B)$$

If $l \in h_i(A)$, then $\exists s \in A$ such that $h_i(s) = l$; since $s \in A \implies s \in A \cup B$, we have $h_i(s) \in h_i(A \cup B)$. If instead $l \in h_i(B)$, then $\exists s \in B$ such that $h_i(s) = l$; since $s \in B \implies s \in A \cup B$, we have $h_i(s) \in h_i(A \cup B)$. \square

13 Randomized min-cut algorithm

Consider the randomized min-cut algorithm discussed in class. We have seen that its probability of success is at least $\frac{1}{\binom{n}{1}}$, where n is the number of its vertices.

- Describe how to implement the algorithm when the graph is represented by adjacency lists, and analyze its running time. In particular, a contraction step can be done in $O(n)$ time.
- A weighted graph has a weight $w(e)$ on each edge e , which is a positive real number. The min-cut in this case is meant to be min-weighted cut, where the sum of the weights in the cut edges is minimum. Describe how to extend the algorithm to weighted graphs, and show that the probability of success is still $geq \frac{1}{\binom{n}{2}}$ [hint: define the weighted degree of a node].
- Show that running the algorithm multiple times independently at random, and taking the minimum among the min-cuts thus produced, the probability of success can be made at least $1 - \frac{1}{nc}$ for a constant $c > 0$ (hence, with high probability).

Contraction algorithm. We assume that the adjacency lists $Adj[x]$ are sorted $\forall x \in V$. We also note that the multigraph is undirected, therefore $v \in Adj[u]$ if and only if $u \in Adj[v]$. We maintain an attribute pe in each edge to store the number of parallel edges.

To contract an edge (u, v) , we have to merge the two adjacency lists $Adj[u]$ and $Adj[v]$ into a single sorted adjacency list $Adj[uv]$ — like the merge procedure in merge sort — ensuring that:

- if the current node in $Adj[u]$ is v , skip the node, since it will not appear in $Adj[uv]$ (do the same with $Adj[v]$ and u);
- if the current nodes are equal to x , add x to $Adj[uv]$ and set $(uv, x).pe = (u, x).pe + (v, x).pe$.

Finally, we replace $Adj[u]$ and $Adj[v]$ with the newly created $Adj[uv]$.

Weighted graph extension. We now extend the above to weighted multi-graph. First we'll define the *min-weighted cut*, the weighted cut where $\min_{|C|>0} C = \min_C \sum w(c)$, that is the sum of the weights determines the weight of the overall cut. We can then define the weighted degree $dg_w(v)$ of a node v as the sum of the weights of its incoming/outgoing edges: $\sum w(e_{v,v'})$. We then reconduct this problem to the non-weighted case: given an edge v with weight $w_v = dg_w(v)$ and split it in w_v edges of weight 1. We then have an equality between the sum of the weights for v is equal to the number of edges (cuts) for v as in the unweighted case. We then apply the error analysis seen in class to obtain $\frac{1}{\binom{n}{2}}$.

Error probability. By the analysis seen in class we have an error probability of

$$1 - \Pr(\text{success}) = 1 - \frac{1}{\binom{n}{1}}$$

if we then run the algorithm some $d \cdot \frac{1}{\binom{n}{2}}$ times, the probability of success becomes

$$1 - \left(1 - \frac{1}{\binom{n}{2}}\right)^{c \cdot \frac{1}{\binom{n}{2}}} \geq 1 - e^{-d}$$

by $d = c \ln(c)$ we have an error probability of $\leq \frac{1}{nc}$.

14 External memory implicit searching

Given a static input array A of N keys in the EMM (external memory or cache-aware model), describe how to organize the keys inside A by suitably permuting them during a preprocessing step, so that any subsequent search of a key requires $O(\log_B N)$ block transfers using just $O(1)$ memory words of auxiliary storage (besides those necessary to store A). Clearly, the CPU complexity should remain $O(\log N)$. Discuss the I/O complexity of the above preprocessing, assuming that it can use $O(N/B)$ blocks of auxiliary storage. (Note that the additional $O(N/B)$ blocks are employed only during the preprocessing; after that, they are discarded as the search is implicit and thus just $O(1)$ words can be employed.)

Solution. The idea is to construct a B-tree, a balanced search tree where each node has B keys. The keys inside a node x divide the interval of keys stored below x in $B + 1$ intervals, therefore each node has $B + 1$ children. We don't store pointers explicitly, the index of the j -th child of a node i is

$$i(B + 1) + B(j + 1) \quad \text{where } 1 \leq j \leq B + 1 \text{ and } 0 \leq i < A.length$$

This is a generalization of the formula for implicit binary heaps, in which $B = 1$.

Assuming that A is sorted, we construct the tree as follows:

1. if $A.length \leq B$ then STOP, since A is the root of the tree;
2. otherwise, select the keys in A to move to the upper level, those whose position i is such that $i \bmod (B + 1) = B$;
3. let L be the keys selected in the previous step, store the remaining $A \setminus L$ keys as leaves, sorted and grouped in blocks of size B ;
4. repeat the process with $A \leftarrow L$.

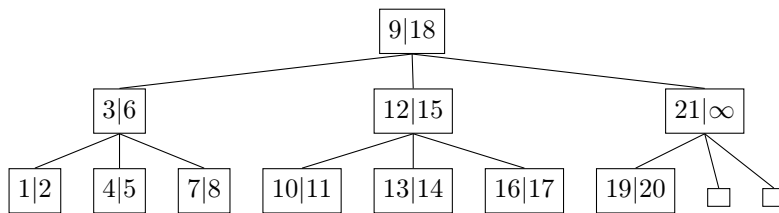
For example, suppose that $B = 2$ and the keys are $1, 2, \dots, 21$. First, we select the keys to move to the upper level (those boxed), while the remaining keys will be the leaves of the tree:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

We repeat the process with the keys selected at the previous iteration (again, the remaining keys form a new level of the tree):

3 6 9 12 15 18 21

In the third iteration we have 2 keys and we can stop, since they can be both stored in a single block that will be the root of the tree:



The tree will be represented in a file in BFS layout:

9 18 3 6 12 15 21 ∞ 1 2 4 5 7 8 10 11 13 14 16 17 19 20

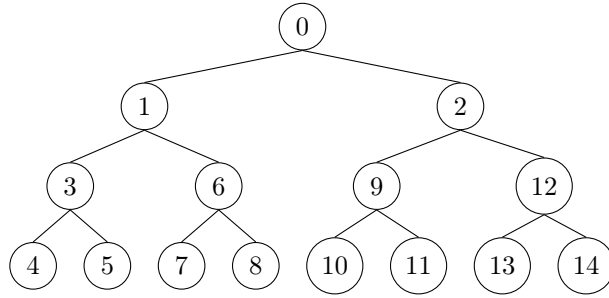
Space and I/O complexity. The construction algorithm copies the permutation of the N keys from A to a new file, and thus the total additional space needed in external memory is $O(N/B)$ blocks. After the preprocessing, A is discarded, hence the auxiliary space is $O(1)$.

The I/O complexity of the step 3 in the first iteration is $O(N/B)$, because we move $m = N - N/(B+1)$ keys in $A \setminus L$, from A to the new file, with $O(m/B) = O(N/B)$ read/write operations. Step 1 requires only $O(1)$ transfers to write the root block to the beginning of the new file. Steps 2 and 4 don't require any transfer from/to external memory, only CPU operations. Subsequent iterations of the algorithm work on smaller portions of A , thus the first iteration dominates with a total cost of $O(N/B)$ I/O operations.

15 Implicit navigation in vEB layout

Consider $N = 2^h - 1$ keys where h is a power of 2, and the implicit cache-oblivious vEB layout of their corresponding complete binary tree, where the keys are suitably permuted and stored in an array of length N without using pointers (as it happens in the classical implicit binary heap but the rule here is different). The root is in the first position of the array. Find a rule that, given the position of the current node, it is possible to locate in the array the positions of its left and right children. Discuss how to apply this layout to obtain (a) a static binary search tree and (b) a heap data structure, discussing the cache complexity.

Solution. We assume that the keys are stored in a zero-indexed array T . Given an index $0 \leq i < T.length$ we will give a rule for computing the index of its children.



First, note a simple property: if $i \bmod 3 = 0$, then the left and the right child of i are stored, respectively, in positions $\text{LEFT}(i) = i + 1$ and $\text{RIGHT}(i) = i + 2$. Observe that the index of the keys that respect this property are those in an even level (the root has level 0).

Before we give a rule for the nodes in an odd level we notice that, in even levels, there is a gap of three indexes between two adjacent *neighbours* (nodes in the same level). Instead, in odd levels: two *siblings* (nodes with the same parent) are stored in adjacent positions; two neighbours that aren't siblings are separated by a gap of one index.

Let k be the largest even power of 2 such that $2^k < i$ (for example, if $i = 8$, such is power is 2). Notice that 2^k is the number of nodes in the k -th level of the tree, and:

- the number of neighbour of i is $2^{k+1} - 1$;
- the index of the first node in the same level of i is $2^k = \alpha$;
- the index of the first node in the same level of i 's children is $2^{k+2} - 1 = \beta$.

Now, since we know the index of the first key in the same level of i 's children, and this level is even, by our previous observation about gaps of three indexes, we can find the left child of i with the following

$$\beta + 3m \quad \text{where } m \text{ is the number of neighbours on the left of } i\text{'s left child}$$

We compute m as twice the difference between i and the index of the first node in the same level minus the number of gaps (that have size one, by the observation on odd levels): $m = 2(i - \alpha - \lfloor \frac{i-\alpha}{3} \rfloor)$.

The final rule to compute the left child of i when $i \bmod 3 \neq 0$ is:

$$\text{LEFT}(i) = 2^{k+2} - 1 + 6 \left(i - 2^k - \left\lfloor \frac{i - 2^k}{3} \right\rfloor \right)$$

and the right child is

$$\text{RIGHT}(i) = \text{LEFT}(i) + 3$$

Binary search tree in vEB layout. We now present a procedure to transform an array of keys S to a tree T , stored in memory in vEB layout, that satisfies the binary search tree property (the key in each node must be greater than all keys stored in the left subtree, and not greater than all keys in the right subtree).

Require: S sorted in ascending order

```

1: function ARRAYTOVEB( $S, i_{\text{vEB}}, l, r$ )
2:   if  $l > r$  then
3:     return
4:    $m \leftarrow \lfloor (l + r)/2 \rfloor$ 
5:    $T[i_{\text{vEB}}] \leftarrow S[m]$  ▷ store the median in the root of the subtree
6:    $l_{\text{vEB}} \leftarrow \text{LEFT}(i_{\text{vEB}})$ 
7:    $r_{\text{vEB}} \leftarrow \text{RIGHT}(i_{\text{vEB}})$ 
8:   ARRAYTOVEB( $S, l_{\text{vEB}}, l, m$ )
9:   ARRAYTOVEB( $S, r_{\text{vEB}}, m + 1, r$ )

```

The recursion starts from the call $\text{ARRAYTOVEB}(S, 0, 0, S.length - 1)$. At the end of the procedure we can binary search in T starting from the root $T[0]$, then traversing the implicit tree with the functions LEFT and RIGHT .

Heap tree in vEB layout. Given any array S sorted in increasing (decreasing) order, the implicit tree with root $S[i]$, children $S[\text{LEFT}(i)]$ and $S[\text{RIGHT}(i)]$, satisfies the min-heap (max-heap) property $\forall i \in [0, S.length - 1]$.

15.1 Formal description of the rule

We define q as the queried index, T the *vEB layout*, h the height of T s.t. $h = 2^l, l > 0$, $e_{l,i}$ as the i^{th} element at level l and A the 1-indexed array. We then show how it can trivially be scaled to a 0-indexed array. Then the following holds:

- **Even powers:** Let k be a level in T s.t. $k \bmod 2 = 0$ s.t. the first element $e_{1,2^k}$ at k : then $e_1 = 2^{2^k}$. Since h is a power of two we'll have the tip of the tree as $1 \mid 2 - 3 \mid 4 - 7 - 10 - 13 \mid \dots \mid$.
- **2-levels stripes:** Let k be a level in T s.t. $k \bmod 2 = 0$ s.t. the first element $e_1 = 2^k$ at k : $e_{1,k} = 2^k$: then $e_{1,k+1} = 2^{k+2}$. Given that $(k, k+1)$ are two subsequent levels, a *vBE* minimal tree T' exists: then T' is populated sequentially with 2^k and 2^{k+1} subsequent indexes: the first $2^k - 1$ of them allow us to get to 2^{k+1} and the remaining $2^{k+2} - 1$ to 2^{k+2} .
- **Odd gaps:** Let $k+1$ be a level in T s.t. $k \bmod 2 = 1$ s.t. the first element $e_1 = 2^{k-1} + 1$ at k : then $e_{1,k+1} = 2^{k+2}$. Given that $(k, k+1)$ are two subsequent levels, a *vBE* minimal tree T' exists: then T' is populated sequentially with 2^k and 2^{k+1} subsequent indexes: the first $2^k - 1$ of them allow us to get to 2^{k+1} and the remaining $2^{k+2} - 1$ to 2^{k+2} .
- **Even gaps:** Let $k : k \bmod 2 = 0$ then $e_{i+1,k} = e_{i,k} + 3$. By the above $e_{i,k}$ is populated in the minimal tree first, and the two children are populated right after it: the two children have subsequent indexes, and the following $e_{i+1,k}$ is populated with $e_{i,k} + 3$ as $e_{i,k} + 1$ and $e_{i,k} + 2$ have been indexed in $k+1$. Also $k : k \bmod 2 = 1$ then $e_{i+1,k} = e_{i,k} + 1$ or $e_{i+1,k} = e_{i,k} + 2$ if there's no gap or a gap, respectively.

We are then able to provide a rule for searching a given index q . We split the analysis on odd and even level cases:

- **Even levels:** By 15.1 we have that any $e_{i,k} = e_{1,k} + 3m, i > 1$. Given $\alpha = 2^k$ we are able to check in constant time if $e_{i,k} - e_{1,k} + 3m \neq 0$, where we fall through the odd-level case, or $e_{i,k} - e_{1,k} + 3m = 0$ where we answer $2^k + m$, as 2^k nodes have already been traversed.
- **Odd levels:** By 15.1 we have that any $e_{i,k+1} = e_{1,k+1} + (1, 2)m, i \geq 1$. Given that the left child of q is in an odd level, we are able to retrieve its value by its offset w.r.t. the start of the $k+2$ level and its right child by adding 1 to it by 15.1. Since the $k+2$ level has two nodes for each of the

nodes in $k + 1$, if we are able to find the offset on level $k + 1$ we are able to search for q 's children with 15.1. We compute q 's left siblings with the following:

$$m = 2 \cdot (i - \alpha - \lfloor \frac{i - \alpha}{3} \rfloor)$$

where α is the number of gaps we skip. We are then able to retrieve

$$\text{left} = 2^{k+2} - 1 + 6 \left(i - 2^k - \left\lfloor \frac{i - 2^k}{3} \right\rfloor \right)$$

and

$$\text{right} = \text{left} + 1$$

16 1-D range query

Describe how to efficiently perform one-dimensional range queries for the data structures described in Problems 14 and 15. Given two keys $k_1 \leq k_2$, a range query asks to report all the keys k such that $k_1 \leq k \leq k_2$. Give an analysis of the cost of the proposed algorithm, asking yourself whether it is output-sensitive, namely, it takes $O(\log_B N + R/B)$ block transfers where R is the number of reported keys.

Solution. Output sensitive algorithms minimize the number of block transfers to an $\alpha = cRB$ in a query, that is no more blocks than necessary are retrieved from memory.

B-tree By our construction a punctual query is answered with $\log_B(N)$ memory transfers. Let us now extend such queries to ranges similarly to the previous exercise on range updates. Given k_1, k_2 interval extremes and B_i the current block we store in memory S , sum of all the keys up until now, and I, J addresses of the block comprising the interval at some level k s.t. k is the highest level containing k_1, k_2 . We assume we are provided with enough space to store also pointers to some, if not all the blocks that comprise k_1, k_2 in order not to traverse them more than one time. If such storage is not available, we scale to pointers to the nodes comprising k_1, k_2 and we restart our search from there if necessary. Given that our tree is a k -ary tree we search through a trivial k -ary search, summing the values we meet in the path if they are in the desired interval. Moreover our construction allows us to assert that given any sub-tree T' an in-order visit grants us a contiguous interval: we are then able to load each of the R'/B blocks in memory and sum them in S . If the interval is completed, then we have our answer in $h' + \frac{R'}{B}$ loads; otherwise we need to go back to the previous highest node in p' where we split the interval and repeat at most $h'' + \frac{R''}{B}$. Since we loaded only blocks whose keys belong to $[k_1, k_2]$ we have $R' + R'' = R$ and $h'' + \frac{R''}{B} + \log_B h' + \frac{R'}{B} = 2\log_B N + \frac{R}{B}$ giving an output-sensitive search algorithm.

vEB layout We've shown previously as *vEB trees* with an height of 2^k store adjacent indexes adjacently, effectively providing an ordered array that we can load with $\frac{R}{B}$ blocks.

A Hogwarts

The Hogwarts School¹ is modeled as a graph $G = (V, E)$ where V is the set of castle's rooms and $E \subseteq V \times V$ is the set of the stairs. Each stair is labelled with the time of appearance and disappearance, and can be walked in both directions, therefore the graph is undirected. The goal is to find, if possible, the minimum amount of time required to go from the first to the last room.

A.1 Solution 1: Preprocessing-then-Dijkstra

Dijkstra is able to find the shortest path in a graph with non-negative weights on its edges. Our main idea is to create a Dijkstra compatible graph through a `NORMALIZE` function, then apply Dijkstra to it in order to find the shortest path. The core of the preprocessing is the `NORMALIZE` function which computes traversal times between nodes at a given time *time*:

```

1: function NORMALIZE(from, to, time):
2:    $t \leftarrow \infty$ 
3:   if  $start[v'] \leq t < end[v']$  then                                     ▷ No waiting time
4:      $t \leftarrow t + 1$ 
5:   else if  $t < start[v']$  then                                           ▷ Waiting time
6:      $t \leftarrow start[v'] + 1$ 
7:   else
8:      $t \leftarrow \infty$                                                  ▷ Available time already expired
9:   return  $t$ 

```

The normalize function is then applied to a node traversal:

A.1.1 Pseudo-code

```

1: create vertex set  $Q$  of unvisited nodes
2: create vertexes set  $E'$  of edges weight
3:  $time \leftarrow 0$                                                          ▷ Initial time for traversal
4:  $edges \leftarrow \text{STAIRS\_OF}(0)$                                        ▷ Get incoming/outgoing edges of the source node
5: function PROCESS(node, time)
6:   if  $edge \in \text{visited\_edges}$  then
7:     return
8:    $traversal\_time \leftarrow \infty$ 
9:   for all  $neighbor \in \text{neighbors\_of\_node}$  do
10:     $traversal\_time \leftarrow \text{TRAVERSAL\_TIME}(node, neighbor, time)$ 
11:     $E'[0][node] \leftarrow traversal\_time$                                 ▷  $E'[i][j]$  holds the weight/traversal
12:                                                         ▷ time for the stair between  $i$  and  $j$ 
13:    for all  $new\_neighbor \in \text{neighbors\_of\_neighbor}$  do
14:      NORMALIZE( $neighbor$ ,  $new\_neighbor$ ,  $traversal\_time$ )
15:   if  $\text{DIJKSTRA}(V, E') = \infty$  then
16:     return -1
17:   else
18:     return  $t$ 

```

Computational cost: $\Theta(n^2)$ if the vertex set in `DIJKSTRA` is implemented as an array. $O(|E| + |V| \log |V|)$ with Fibonacci heap.

A.2 Solution 2: HogwartsDijkstra

```

1: function HOGWARTSDIJKSTRA( $G$ ):
2:   create vertex set  $Q$  of unvisited nodes

```

¹http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformatica/alg2/algo2_16/hogwarts.pdf

```

3:  for all vertex  $v \in V$  do                                     ▷ initialization
4:       $time[v] \leftarrow \infty$                                 ▷ unknown time from source to  $v$ 
5:      add  $v$  to  $Q$                                              ▷ all nodes initially in  $Q$ 
6:   $time[0] \leftarrow 0$                                          ▷ time from source to source
7:  while  $Q \neq \emptyset$  do
8:       $u \leftarrow x \in Q$  with  $\min\{time[x]\}$ 
9:      remove  $u$  from  $Q$ 
10:     for all neighbor  $v$  of  $u$  do:
11:         if  $time[u] \leq appear[v]$  then
12:              $alt \leftarrow appear[v] + 1$                      ▷ wait the appearance of the stair
13:         else if  $time[u] < disappear[v]$  then
14:              $alt \leftarrow time[u] + 1$                        ▷ use the stair
15:         else
16:              $alt \leftarrow \infty$                              ▷ the stair has already disappeared
17:         if  $alt < time[v]$  then
18:              $time[v] \leftarrow alt$                            ▷ a quicker path to  $v$  has been found
19:  return  $time[N] - 1$ 

```

Computational cost. See the previous section.

A.3 Solution 3: BFS-like traversal

```

1: function REACH( $N, M, A[], B[], appear[], disappear[]$ )
2:     for  $i = 0$  to  $M - 1$  do
3:          $edges\_A[i].push\_back(make\_pair(i, B[i]))$ 
4:          $edges\_B[i].push\_back(make\_pair(i, A[i]))$ 
5:     for  $i = 0$  to  $N - 1$  do
6:          $done\_i[i] \leftarrow false$ 
7:          $distance\_i[i] \leftarrow \infty$ 
8:      $reached\_0[0].push\_back(0)$ 
9:      $distance\_0[0] \leftarrow 0$ 
10:    for  $t = 0$  to  $MAX\_TIME$  do
11:        for all  $v \in reached\_t[t]$  do
12:            if not  $done\_i[v]$  then
13:                for all  $edge \in edges\_v[v]$  do
14:                     $staircase \leftarrow edge.first$ 
15:                     $neighbor \leftarrow edge.second$ 
16:                     $time \leftarrow \max(distance\_v[v], appear[staircase]) + 1$ 
17:                    if not  $done\_i[neighbor]$ 
18:                        and  $distance\_v[v] < disappear[staircase]$ 
19:                        and  $time < distance\_i[neighbor]$  then
20:                         $distance\_i[neighbor] \leftarrow time$ 
21:                         $reached\_t[time].push\_back(neighbor)$ 
22:                     $done\_i[v] \leftarrow true$ 
23:    return  $(distance\_i[N - 1] = \infty) ? -1 : distance\_i[N - 1]$ 

```

Computational cost: $O(m + MAX_TIME)$.

B Paletta

Paletta ordering² is a peculiar ordering technique: given a 3-tuple of elements, paletta takes the central element as pivot and swaps the two elements right before and next to it. To make an example:

$$(3, 2, 1) \xrightarrow{\text{paletta}} (1, 2, 3)$$

We now want to develop an algorithm to order any array through paletta ordering with the minimal number of swaps. You should see as not every array can be ordered (e.g. $[1, 3, 2]$).

Solution. We should note that the following properties hold:

1. Every element can be a pivot, but the first and the last one, as they have respectively no elements before and after them.
2. Every element can be swapped as many times as necessary, but only with elements of the same 2-remainder (numbers in even positions can only be swapped with numbers in even positions, the same holds for odd indexes). More formally, if n is the size of the array A we want to sort, $i, j \in [1, n-2]$, $A[i]$ can be swapped with $A[j]$ if and only if $i \equiv j \pmod{2}$.
3. The least number of swaps does not backtrack any element. Formally, let k be the minimal number of swaps applied to an array, backtracks included. By hypothesis, k is minimal, but at least m , $m > 0$ backtrack swaps have been operated, therefore we found a $k' = k - m : k' < k$, a new minimal number of swaps: contradiction.

Given item 2, we can split our array in two, even and odd numbers, and order them counting the swaps. In our example we'll use *mergesort*, as it runs in $O(n \log n)$, does backtrack elements, and is very well-known. Clearly, given an array, a swap happens when an element is pushed back, pulling the one between its new position and the old one ahead: we can map this behaviour in the merge routine of mergesort: the array merged is able to push back elements from its right pointer to the new array, moving them back of $(m-i) + (j-m)$ positions, where m is the dimension of the current two sub-arrays to merge. Provided that our edited version of mergesort ran successfully on both the even-index and odd-index, we now need to verify if by merging them we obtain an ordered array. Intuitively, the merged array will start with the first element of the even-index arrays, followed by the first of the odd-index array, followed by the second of the even-index array, and so on. To check for these elements is pretty trivial and can be done in linear time. Follows the pseudo-code for the edited version and SNAKE_CHECK function:

```
1: function MERGE_WITH_PALETTA(left, right, k):  
2:   ... ▷ merge instructions  
3:   if right > left then  
4:     paletta_count ← paletta_count + 1  
5:     ...  
1: function SNAKE_CHECK  
2:   even, odd ← 0  
3:   for ;even, odd < N; even = even + 1, odd = odd + 1 do  
4:     if a[even] > a[odd] then  
5:       return -1  
6:   return paletta_count
```

Computational cost: $\Theta(n \log n)$.

²http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformatica/alg2/algo2_16/paletta.pdf

References

- [1] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.
- [2] Graham Cormode. “Sketch techniques for approximate query processing”. In: *Synposes for Approximate Query Processing: Samples, Histograms, Wavelets and Sketches, Foundations and Trends in Databases*. NOW publishers. 2011. URL: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.374.2889>.
- [3] Graham Cormode and S. Muthukrishnan. “An Improved Data Stream Summary: The Count-min Sketch and Its Applications”. In: *J. Algorithms* 55.1 (Apr. 2005), pp. 58–75. ISSN: 0196-6774. URL: <http://dx.doi.org/10.1016/j.jalgor.2003.12.001>.
- [4] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.