Advanced Algorithms Problems and Solutions

Mattia Setzu Giorgio Vinciguerra

October 2016

Contents

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 $\Sigma_{k=1}^{j}(C[k])$.

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).]

1.1 Solution 1: Segment tree lazy a-b sums

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

$$s' : [0, n-1] \to \mathbb{N} \tag{1}$$

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:

$$l : \mathbb{N} \to (\mathbb{N} \cup \{\epsilon\}, \mathbb{N}) \tag{2}$$

to keep track of our lazy sums:

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

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 ∈ I.left, i ∉ I.right then                    ▷ Lazy on left child
5:          lazy(I) ← False
6:          QUERY(I.left, i, sum + I.sum)
7:      if lazy(I), i ∈ I.right, i ∉ I.left then                    ▷ Lazy on right child
8:          lazy(I) ← False
9:          QUERY(I.right, i, sum + I.sum)
10:     if lazy(I), i ∈ I.right, i ∈ I.left then                    ▷ Lazy on both
11:         lazy(I) ← False
12:         QUERY(I.right, i, j, sum + I.sum) + QUERY(I.left, i, j, sum + I.sum)
13:     if !lazy(I), i ∈ I.left then                                ▷ Not lazy on left child
14:         QUERY(I.left, i, sum)
15:     if !lazy(I), i ∈ I.right then                               ▷ Not lazy on right child
16:         QUERY(I.right, i, sum)
17:     if !lazy(I), i ∈ I.right, i ∈ 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 probability that the expected depth of a node exceeds $2 \ln n$ is small for any given constant $c > 1$.

Chernoff's bounds.

2.1 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 = 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 balanced, T' 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 costantly force, at each level, one subtree to disappear. Therefore, its level(s) has to be necessarly 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 2.1k-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 2.1, 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:

$$P(x_k) = \frac{1}{n - l} \tag{4}$$

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

$$P(x_k) = \Sigma_{k=1}^{l-1}(\frac{1}{n-l+1}) \tag{5}$$

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.

2.2 Upper bound

By hypothesis,

$$\mathbb{E}[d(x) > 2c\ln(n)] <<< 1 \tag{6}$$

By definition the ancestor of a node i are indipendent random variables, and we can apply the Chernoff bounds over the set $x : d(x) >= 2\ln(n)$ 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

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

2.3 Proof with indicator variable

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

Proof. Let z_m the mth 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:

$$X = \sum_{\substack{j=1 \\ j \neq i}}^{n} X_{ij} \tag{7}$$

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 (7), 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 it's descendants. Since (7) 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 $j = 1, \ldots, n$ to $i = 1, \ldots, n$. □

3 Karp-Rabin fingerprinting on strings

Given a string $S : |S| = n$, and two positions $0 \leq i < j \leq (n-1)$, the longest common extension $lceS(i, j)$ is the length of the maximal run of matching characters from those positions, namely: if $S[i]6 = S[j]$ then $lceS(i, j) = 0$; otherwise, $lceS(i, j) = \max l \geq 1 : S[i...i + l - 1] = S[j...j + i - 1]$. For example, if S = abracadabra, then $lceS(1, 2) = 0$, $lceS(0, 3) = 1$, and $lceS(0, 7) = 4$. Given S in advance for preprocessing, build a data structure for S based on the Karp-Rabin fingerprinting, in $O(n \ln(n))$ time, so that it supports subsequent

- $lceS(i, j)$: it computes the longest common extension at positions i
- $equals(i, j, c)$: it checks if $S[i...i + ` - 1] = S[j...j + ` - 1]$ 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 onetime preprocessing is performed, and then many online queries are to be answered on the fly.]

3.1 Solution 1: Cumulative shift

3.1.1 Construction

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 computating 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:

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

3.1.2 equals(i, j, l)

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}
```

3.1.3 lce(i, j)

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 = EQUALS(i, j)
3:     if eq then return l
4:     else if ¬ EQUALS((j − i)/2, (n − j)/2, l) then return EQUALS((j − i)/2, (n − j)/2)
5:     else(j − i)/2+ 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}
```

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 BS 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 BR 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
R (difference)? What is the probability of error?

4.1 $k \in S$

Trivially for $B_s[h(k)] = 0$ we can answer FALSE with $\mathbb{P}[error] = 0$. 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 $k \in S : h(k) = 1$ for some set $S \in \mathcal{P}(S)$. We can then denote the sets $cl_U := k_U : k_U \in U, h(k) = i, cl_S := k_S : k_S \in S, h(k) = i$; it is trivial to show that

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

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

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

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

Given our estimate of the collision list, we can now compute an estimate of the error probability: by 4.1, 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

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

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 $c > 1, |S| \leq \frac{m}{c}$:

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

Therefore

$$e \leq 1 - \frac{1}{m^{|S|}} \leq \frac{1}{c} \tag{10}$$

4.2 Expected number of 1 in B_s

We define a random variable X_k:

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

and one over the ones in B_s, X:

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

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

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

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

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

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 sufficently 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 $a+pk \bmod p = c \forall k \in K, K_N = k_i : k_i < N$

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

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:

$$(ax_1 + b) \bmod p = d$$
$$(ax_2 + b) \bmod p = e$$

By the Chinese reminder 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):

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

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

$$r = (ax_1 + b) \bmod p$$
$$s = (ax_2 + b) \bmod p$$

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):

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

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:

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

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

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

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

□

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 $O(polylog(n))$ bits (i.e. $O(log^c n)$ for any fixed constant $c > 0$): in other words, only $b = O(k \cdot polylog(n))$ 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 configuration with at least two different streams. In fact, the number of configurations of memory for this class of algorithms is bounded because of b and, by the pigeonhole principle, there exist at least two streams that cause the algorithm to transition the same configuration.

There are 2^b possible configurations of the memory. If we use an alphabet Σ such that $|\Sigma| = 2^b$, then the algorithm would need $log_2|\Sigma| = b$ bits to distinguish each symbol. Intuitively, given such alphabet, if we feed the stream with streams $s \in \Sigma^*$, then the algorithm can neither store (any representations of) the elements of the stream without loosing information, nor information about the stream, such as counters. We therefore create a class of streams:

$$\mathcal{S}_\Sigma = \{s \mid s \text{ contains all the symbols in } \Sigma \text{ and there is a tie on frequencies}\}$$

Note that $\forall s \in \mathcal{S}_\Sigma$, $|s| \geq |\Sigma|$, hence we can apply the pigeonhole principle: suppose that the same configuration is reached in two different executions of the algorithm with the two streams $\alpha \in \mathcal{S}_{\Sigma_1}$ and $\beta \in \mathcal{S}_{\Sigma_2}$, where $|\Sigma_1| = |\Sigma_2| = 2^b$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$. When we will emit $a \in \Sigma_1$ 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 the same configuration the answer will be the same for each stream. But $\mathcal{A}$ will be wrong for at least one of the two streams, because they have different alphabets.

stream $\alpha \in \mathcal{S}_{\Sigma_1}$	...	d	e	f	g	h
stream $\beta \in \mathcal{S}_{\Sigma_2}$	...	4	5	6	7	8

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.

7.1 Solution 1: Up & Down counter

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, $\Sigma_{i=0, i \neq 1}^{\frac{n}{2}}(C(j^i) < C(j^1)$ by 7.1. Though trivial, by combining it with 7.1 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 $\exists j_o \neq j^1 : C(j_o)$ in $S' > C(j^1$, $\exists j_o : C(j_o)$ 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

$$e - k - 1 \text{ in } S', \frac{n}{2} + 1 - (e - k - 1) \text{ in } S'' \tag{13}$$

  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 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 7.1 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. Note that C holds only one element, for simplicity we'll denote its entries as a map in a Scala-like syntax.

```
1: function UP_AND_DOWN(S, C):
2:     k ← S.next()
3:     if ∃C(k) then
4:         C ← {k => C(k) + 1}                          ▷ Increment current dominant element.
5:     else if C(k) == 0 then
6:         C ← {k => 1}          ▷ k is no more dominant, swap it with new element and initialize.
7:     else
8:         C ← {k => C(k) − 1}                          ▷ k is still dominant, but the stream.
9:     return C
```

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 $c = \frac{e}{\varepsilon}$, that we use to approximate queries on F.
- r hash functions $h_1, \ldots, h_r$, chosen uniformly at random from a pairwise-independent family, that maps from $\{1, \ldots, n\}$ to $\{1, \ldots, c\}$.

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:

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

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. $\mathbb{P}\left[\tilde{F}[i] > F[i] + \varepsilon \|F\|\right] \leq \delta$.

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

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

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

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,

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

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

$$\mathbb{E}\left[|X_{i,j}|\right] = \mathbb{E}\left[\sum_{k=1}^{n} I_{i,j,k} |F[k]|\right] \leq \sum_{k=1}^{n} |F[k]| \mathbb{E}\left[I_{i,j,k}\right] \leq \frac{\varepsilon}{e}\|F\| \tag{14}$$

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

$$\begin{aligned}
&\mathbb{P}\left[|T[i, h_j(i)] - F[i]| \geq 3\varepsilon\|F\|\right] \\
&= \mathbb{P}\left[|F[i] + X_{i,j} - F[i]| \geq 3\varepsilon\|F\|\right] && \text{by construction of } T \\
&= \mathbb{P}\left[|X_{i,j}| \geq 3\varepsilon\|F\|\right] \\
&\leq \frac{\mathbb{E}\left[|X_{i,j}|\right]}{3\varepsilon\|F\|} && \text{by the Markov inequality} \\
&\leq \frac{\varepsilon\|F\|}{e} \frac{1}{3\varepsilon\|F\|} && \text{by (14)} \\
&= \frac{1}{3e} < \frac{1}{8}
\end{aligned}$$

Applying Chernoff bounds... □

8.2 Second solution

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

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

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: $\mathbb{P}\left[\tilde{F}[i] > F[i] + \epsilon\|F\|\right]$. 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 $\frac{r}{2}$:

$$\mathbb{P}\left[\tilde{F}[i] \in F[i] + \epsilon\|F\|\right] =$$
$$\mathbb{P}\left[\tilde{F}[i] \geq F[i] - |\epsilon|\|F\| \wedge \tilde{F}[i] \leq F[i] + |\epsilon|\|F\|\right] =$$
$$\mathbb{P}\left[\tilde{F}[i] \geq F[i] - |\epsilon|\|F\|\right) \mathbb{P}\left[(\right] \tilde{F}[i] \leq F[i] + |\epsilon|\|F\|\right]$$

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

$$p_0 = \prod^{\frac{r}{2}} \mathbb{P}\left[|Xji| \geq -|\epsilon|\|F\|\right] =$$
$$= -\frac{1}{2^{\frac{r}{2}}} = -\delta^{\frac{1}{2}}$$

Now for p_1:

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

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 $c_1, c_2, ..., c_t$. 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 suffices 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 it's 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 $O(\frac{1}{\varepsilon} \log n \log \frac{1}{\delta})$, because each sketch requires $O(\frac{e}{\varepsilon} \ln \frac{1}{\delta})$ 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:

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

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 $\mathbb{E}[I_{i,j,k}] \leq \frac{\varepsilon}{e}$ and

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

A range query $\tilde{F}[l..r]$ is assembled by $t \leq 2 \log n$ point queries on dyadic intervals $[l_1, r_1], [l_2, r_2], \ldots, [l_t, r_t]$. 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

$$\mathbb{E}\left[X^j\right] = \mathbb{E}\left[\sum_{s=1}^{t} X_{i,j}\right] = \sum_{s=1}^{t} \mathbb{E}[X_{i,j}] \leq (2 \log n)\frac{\varepsilon}{e}\|F\| \tag{15}$$

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

$$\tilde{F}^j[l..r] = \sum_{s=1}^{t} \tilde{F}^j[l_s..r_s] = \sum_{s=1}^{t} (F[l_s..r_s] + X_{i,j}) = F[l..r] + X^j \tag{16}$$

Now, for the j-th estimation we can compute

$$\begin{aligned}
&\mathbb{P}\left[\tilde{F}^j[l..r] > F[l..r] + 2\varepsilon \log n \|F\|\right] \\
&= \mathbb{P}\left[X^j > 2\varepsilon \log n \|F\|\right] && \text{because of (16)} \\
&\leq \frac{\mathbb{E}\left[X^j\right]}{2\varepsilon \log n \|F\|} && \text{by the Markov inequality} \\
&\leq \frac{2\varepsilon \log n \|F\|}{e} \frac{1}{2\varepsilon \log n \|F\|} && \text{because of (15)} \\
&= e^{-1}
\end{aligned}$$

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

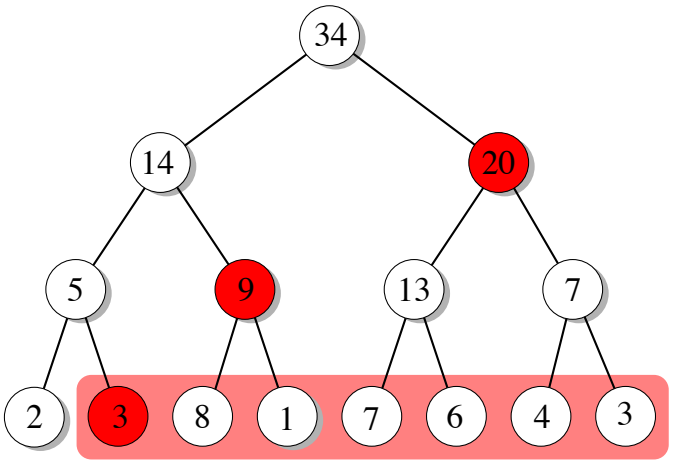

Figure 1: A hierarchy of dyadic ranges. The leaves are the ranges of length 2^0, while the root corresponds to the single range of length 2^3. Each level of the tree can be seen as a CM sketch table. To estimate $\sum_{k=2}^{8} F[k]$, the range $[2, 8]$ is decomposed into dyadic ranges $[2, 2], [3, 4], [5, 8]$. 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 $h : U \to [m]$ 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 $h_j : U \to [m_j]$ 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 jth 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 $b \to \infty$.

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:

$$L = \overbrace{00\ldots0}^{n_0^2 \text{ times}}1\overbrace{00\ldots0}^{n_1^2 \text{ times}}1\cdots\overbrace{00\ldots0}^{n_{n-1}^2 \text{ times}}1$$

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 jth 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 bucket, 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 $s : U \to [m]$ 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.

11.1 Solution 1

11.2 Error probability

By swapping to a signature function on a $\frac{1}{f}$ table we get back to our classic definition of hash function with a one-sided error of $\frac{1}{m}$ where $m = \frac{1}{f} \implies \Pr(error) = \frac{1}{\frac{1}{f}} = f$

11.2.1 Bits count comparison

Our solution uses roughly $17n + o(n) \approx \frac{17}{f} + o(\frac{1}{f})$ bits per key, while Bloom filters clock around $1.44 \log_2(\frac{1}{f})$. We add an hash table T_s to store the signatures $s(k)$ with size $m \cdot \log_2(m) = \frac{1}{f} \cdot \log_2(m)$ bits. We try and analyse these two functions $g_{bloom}(n) < g_{perf\_hash}(x)$ and verify whether or not they meet:

$$\log_2(1/f) + 17 < 1.44 \log_2(1/f)$$
$$f < \frac{1}{2^{\frac{17}{0.44}}}$$

That is, bloom filters are more performing for f over $2.34 \cdot 10^{-12}$.

12 MinHash sketches

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

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

with each set X in the given data collection, where $h_1, h_2, \ldots, h_k$ 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

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

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

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

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 $h_i(s) \in h_i(A \cup B)$. □

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).

13.1 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 $Adj[uv]$.

13.2 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 $\frac{1}{\binom{n}{2}}$.

13.3 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 $\leq \frac{1}{n^c}$.

A Hogwarts

The Hogwarts School[1] 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 ← ∞
3:     if start[v'] ≤ t < end[v'] then                    ▷ No waiting time
4:         t ← t + 1
5:     else if t < start[v'] then                         ▷ Waiting time
6:         t ← start[v'] + 1
7:     else
8:         t ← ∞                                          ▷ 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 ← 0                                               ▷ Initial time for traversal
4: edges ← STAIRS_OF(0)                                   ▷ Get incoming/outgoing edges of the source node
5: function PROCESS(node, time)
6:     if edge ∈ visited_edges then
7:         return
8:     traversal_time ← ∞
9:     for all neighbor ∈ neighbors_of_node do
10:        traversal_time ← TRAVERSAL_TIME(node, neighbor, time)
11:        E'[0][node] ← traversal_time                   ▷ E'[i][j] holds the weight/traversal
12:                                                      ▷ time for the stair between i and j
13:        for all new_neighbor ∈ neighbors_of_neighbor do
14:            NORMALIZE(neighbor, new_neighbor, traversal_time)
15:    if DIJKSTRA(V, E') = ∞ 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
```

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

```
    for all vertex v ∈ V do                                          ▷ initialization
        time[v] ← ∞                                                  ▷ unknown time from source to v
        add v to Q                                                   ▷ all nodes initially in Q
    time[0] ← 0                                                      ▷ time from source to source
    while Q ≠ ∅ do
        u ← x ∈ Q with min{time[x]}
        remove u from Q
        for all neighbor v of u do:
            if time[u] ≤ appear[v] then
                alt ← appear[v] + 1                                  ▷ wait the appearance of the stair
            else if time[u] < disappear[v] then
                alt ← time[u] + 1                                    ▷ use the stair
            else
                alt ← ∞                                              ▷ the stair has already disappeared
            if alt < time[v] then
                time[v] ← alt                                        ▷ a quicker path to v has been found
    return time[|N| − 1]
```

Computational cost. See the previous section.

A.3 Solution 3: BFS-like traversal

```
function REACH(N, M, A[], B[], appear[], disappear[])
    for i = 0 to M − 1 do
        edges_[A[i]].push_back(make_pair(i, B[i]))
        edges_[B[i]].push_back(make_pair(i, A[i]))
    for i = 0 to N − 1 do
        done_[i] ← false
        distance_[i] ← ∞
    reached_[0].push_back(0)
    distance_[0] ← 0
    for t = 0 to MAX_TIME do
        for all v ∈ reached_[t] do
            if not done_[v] then
                for all edge ∈ edges_[v] do
                    staircase ← edge.first
                    neighbor ← edge.second
                    time ← max(distance_[v], appear[staircase]) + 1
                    if not done_[neighbor]
                                                          and distance_[v] < disappear[staircase]
                                                            and time < distance_[neighbor] then
                        distance_[neighbor] ← time
                        reached_[time].push_back(neighbor)
                done_[v] ← true
    return (distance_[N − 1] = ∞)? − 1 : distance_[N − 1]
```

Computational cost: $O(m + MAX\_TIME)$.

B Paletta

Paletta ordering[2] 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:

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

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]).

B.1 Solution 1: Split and count-inversions

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

[2] http://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.