Advanced Algorithms Problems and Solutions

Mattia Setzu Giorgio Vinciguerra

October 2016

Contents

1 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.

1.1 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:

1.1.1 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
```

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

```
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.

1.2 Solution 2: HogwartsDijkstra

```
1: function HogwartsDijkstra(G):
2:     create vertex set Q of unvisited nodes
3:     for all vertex v ∈ V do                                   ▷ initialization
4:         time[v] ← ∞                              ▷ unknown time from source to v
5:         add v to Q                                    ▷ all nodes initially in Q
6:     time[0] ← 0                                      ▷ time from source to source
7:     while Q ≠ ∅ do
8:         u ← x ∈ Q with min{time[x]}
9:         remove u from Q
10:        for all neighbor v of u do:
11:            if time[u] ≤ appear[v] then
12:                alt ← appear[v] + 1          ▷ wait the appearance of the stair
13:            else if time[u] < disappear[v] then
14:                alt ← time[u] + 1                                ▷ use the stair
15:            else
16:                alt ← ∞                  ▷ the stair has already disappeared
17:            if alt < time[v] then
18:                time[v] ← alt            ▷ a quicker path to v has been found
19:    return time[|N| − 1]
```

Computational cost. See the previous section.

1.3 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] ← false
7:         distance_[i] ← ∞
8:     reached_[0].push_back(0)
9:     distance_[0] ← 0
```

```
10:    for t = 0 to MAX_TIME do
11:        for all v ∈ reached_[t] do
12:            if not done_[v] then
13:                for all edge ∈ edges_[v] do
14:                    staircase ← edge.first
15:                    neighbor ← edge.second
16:                    time ← max(distance_[v], appear[staircase]) + 1
17:                    if not done_[neighbor]
18:                                          and distance_[v] < disappear[staircase]
19:                                            and time < distance_[neighbor] then
20:                        distance_[neighbor] ← time
21:                        reached_[time].push_back(neighbor)
22:                done_[v] ← true
23:    return (distance_[N − 1] = ∞)? − 1 : distance_[N − 1]
```

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

2 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]).

2.1 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 $i \equiv j \pmod 2$.

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

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

3 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).]

3.1 Solution 1: Fenwick 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
```

```
        QUERY(I.left, i, sum + I.sum)
    if lazy(I), i ∈ I.right, i ∉ I.left then                ▷ Lazy on right child
        lazy(I) ← False
        QUERY(I.right, i, sum + I.sum)
    if lazy(I), i ∈ I.right, i ∈ I.left then                ▷ Lazy on both
        lazy(I) ← False
        QUERY(I.right, i, j, sum + I.sum) + QUERY(I.left, i, j, sum + I.sum)
    if !lazy(I), i ∈ I.left then                            ▷ Not lazy on left child
        QUERY(I.left, i, sum)
    if !lazy(I), i ∈ I.right then                           ▷ Not lazy on right child
        QUERY(I.right, i, sum)
    if !lazy(I), i ∈ I.right, i ∈ I.left then               ▷ Not lazy both
        SUM(I.right, i, sum)
function SUM(I, i, j, sum):
    if I.size = 1 then                                      ▷ Return
        return I.sum + sum
    if lazy(I), i ∈ I.left, i ∉ I.right then                ▷ Lazy on left
        lazy(I) ← False
        SUM(I.left, i, j, sum + I.sum)
    if lazy(I), i ∈ I.right, i ∉ I.left then                ▷ Lazy on right
        lazy(I) ← False
        SUM(I.right, i, j, sum + I.sum)
    if lazy(I), i ∈ I.right, i ∈ I.left then                ▷ Lazy on both
        lazy(I) ← False
        SUM(I.right, i, j, sum + I.sum) + SUM(I.left, i, j, sum + I.sum)
    if !lazy(I), i ∈ I.left then                            ▷ Not lazy on both
        SUM(I.left, i, sum)
    if !lazy(I), i ∈ I.right then                           ▷ Not lazy on both
        SUM(I.right, i, sum)
    if !lazy(I), i ∈ I.right, i ∈ I.left then               ▷ Not lazy on both
        SUM(I.right, i, sum)
function UPDATE(I, i, j, k):
    if I.size = 1 then                                      ▷ Return
        return I.val ← I.val + update
        return I.val+ = update
    if lazy(I), i ∈ I.left, i ∉ I.right then                ▷ Lazy on left
        lazy(I.left) ← True
        I.left.val ← k
    if lazy(I), i ∈ I.right, i ∉ I.left then                ▷ Lazy on right
        lazy(I.right) ← True
        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)
```

4 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.

4.1 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 4.1k-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 4.1, 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.

4.2 Upper bound

By hypothesis,

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

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.

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

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

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

4.3 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:

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

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 $i = 1, \ldots, n$. □

5 Karp-Rabin

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.]

5.1 Solution 1: Cumulative shift

5.1.1 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

5.1.2 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}
```

5.1.3 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** $\neg$ 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}

6 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?

6.1 $k \in S$

Trivially for $B_s[h(k)] = 0$ we can answer FALSE with $\mathsf{P}(error) = 0$. 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 $\mathsf{E}[X_k] \approx \frac{1}{m} \forall k \in S$, that is

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

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

Given our estimate of the collision list, we can now compute an estimate of the error probability: by 6.1, 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

$$1 - \mathsf{P}(\text{good answer}) = 1 - (1 - \frac{1}{m})^{|S|} \tag{8}$$

7 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

$$\mathsf{P}(h(x_1 = c_1), h(x_2 = c_2)) = \mathsf{P}(h(x_1 = c_1)) * \mathsf{P}(h(x_2 = c_2)) \tag{9}$$

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

7.1 Equal probability

By linear algebra, $ak \mod p = c \forall k \in \mathbb{N}$. If we were to cap $ak \mod p = c \forall k \in K, K_N = k_i : k_i < N$

$$\mathsf{P}(h(x_i = c_i)) = \frac{1}{m^2} = \mathsf{P}(h(x_1 = c_1), h(x_2 = c_2)) \tag{10}$$

Given x_i we define as $m_i = (ax_i + b) \mod p$; since $(ax_i + b)$ is a *linear transformation* m_i is unique. It follows trivially that $\mathsf{P}((ax_i + b) = d) = \frac{1}{p}$. Now, the same goes for x_1, x_2:

$$(ax_1 + b) \equiv d \tag{11}$$

$$(ax_2 + b) \equiv e \tag{12}$$

By the Chinese reminder theorem the above system has only one solution, therefore $\mathsf{P}((ax_1 + b) = d) = \frac{1}{p}$ and e become independent of d. Having

$$\mathsf{P}(h(x_1 = c_1), h(x_2 = c_2)) = \mathsf{P}(h(x_1 = c_1)) * \mathsf{P}(h(x_2 = c_2)) = \frac{1}{p} * \frac{1}{p} = \frac{1}{p^2} \tag{13}$$

Which, since $m + 1 \leq p \leq 2m$, proves the assumption under the said bound.

8 Deterministic data streaming

Consider a stream of n items, where items can appear more than once in the stream. The problem is to find the most frequently appearing item in the stream (where ties broken arbitrarily if more than one item satisfies the latter). Suppose that only k items can be stored, one item per memory cell, where the available storage is $k + O(1)$ memory cells. Show that the problem cannot be solved deterministically under the following rules: the algorithm can access only $O(\log^c(n))$ bits for each of the k items that it can store, and can read the next item of the stream; you, the adversarty, have access to all the stream, and the content of the k items stored by the algorithm, and can decide what is the next item that the algorithm reads (please note that you cannot change the past, namely, the items already read by the algorithm).
Hint: it is an adversarial argument based on the k items chosen by the hypothetical determinist streaming algorithm, and the fact that there can be a tie on $> k$ items till the last minute.

8.1 Solution 1: Max replacement

Let us first analyse the problem without the first constraint. Let S be the given stream, $i \in [0, n-1]$ be an iteration in the stream read, $C_{k+O(1)}$ the map $object \in S \to c \in [0, n-1]$, k the size of $C_{k+O(1)}$. We'll denote $C_{k+O(1)}$ with C from now on. We'll also denote j^i as the i^{th} most frequent element.
Suppose that at i, $C = \{j_0 \to c, j_1 \to c, \ldots, j_{k-1} \to c\}$: as by hypothesis, let us be in control of the stream and read $j_o : j_o \notin C$. We can now pick one of two choices: either insert $j_o \to 1$ in C by swapping it with some $j_k \in C$, or ignore it. In the former case, we might end up replacing the most frequent object, making our final solution wrong (note that a $j_{k'} \to c$ exist in C s.t. its value at the end of the read might be greater than $C(j_k) - c$). Given that we are in control of the stream, we can always pick the first and second most frequent objects and make them appear in such a way that we replace the most frequent item with the second most, namely $j^1 and j^2$. We first feed the stream the objects in sequence s.t. we have a

$$j_0, j_0, \ldots, j_0, j_1, j_1, \ldots, j_1, \ldots j_{k-1}, j_{k-1}, \ldots, j_{k-1} \tag{14}$$

where each substream j_i has length $l = \min$(frequence of every item), and one of j_i is the most frequent item. We then proceed to feed $j_o \notin C$ until one of them reaches the same counter of any of the j_i: the algorithm will be forced to replace one of the $j_i \in C$, without knowing which of them is j^1. By postponing this to the m^{th} iteration where $m = n - (C(j^1) - C(j^2))$ we are able to programmatically make the algorithm effectively return, in the best case scenario, the j^2 object, making the algorithm non-deterministic. Let us now show how the problem changes with the other constriction: suppose that each element $j_i \in S$ has m bits length: now if $\log^c(n) < m$ we might not able to tell two j_i, j_k apart, making the algorithm useless also for $k = n$. Instead, if we are able to tell them apart

for a $\log^c(n)$, we fall back in the previous case. Finally, if $\log^c(n) > m$, we overflow.

9 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.

9.1 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 9.1. Though trivial, by combining it with 9.1 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

$$e - k - 1 \text{ in } S', \frac{n}{2} + 1 - (e - k - 1) \text{ in } S'' \tag{15}$$

  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 9.1 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: $k \leftarrow S.next()$
3: **if** $\exists C(k)$ **then**
4: $C \leftarrow \{k => C(k) + 1\}$ ▷ Increment current dominant element.
5: **else if** $C(k) == 0$ **then**
6: $C \leftarrow \{k => 1\}$ ▷ k is no more dominant, swap it with new element and initialize.
7: **else**
8: $C \leftarrow \{k => C(k) - 1\}$ ▷ k is still dominant, but the stream.
9: **return** C