Advanced Algorithms: Problems and Solutions

Mattia Setzu Giorgio Vinciguerra

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 $\sum_{k=1}^{j} C[k]$.

Design a data structure that uses $O(n)$ space, takes $O(n \log n)$ construction time, and implements each operation above in $O(\log n)$ time. Note that $query(i) = sum(i, i)$ but it helps to reason.

[Hint: For the general case, use the segment tree seen in class, which uses $O(n \log n)$ space: prove that its space is actually $O(n)$ when it is employed for this problem.]

1.1 First solution

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

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

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

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

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
```

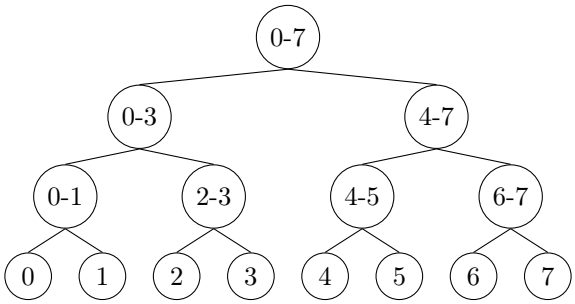
```
18:         SUM(I.right, i, sum)
 1: function SUM(I, i, j, sum):
 2:     if I.size = 1 then                                              ▷ Return
 3:         return I.sum + sum
 4:     if lazy(I), i ∈ I.left, i ∉ I.right then                        ▷ Lazy on left
 5:         lazy(I) ← False
 6:         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)
```

1.2 Second solution

We use a segment binary tree T_I over the interval $I = [0, n-1]$, that is, a tree whose leaves are the points in I and the parent of two nodes is the union of their interval. More formally, if $x.interval$ denotes the attribute $interval$ of the node x,

1. $l.interval \cup r.interval = n.interval \iff n$ is the parent of l and r;
2. if x and y are leaves, then $x.interval \cap y.interval = \emptyset \wedge |x.interval| = |y.interval| = 1$.

For example, the segment tree for $I = [0, 7]$ is the following:



We associate with each node x of T_I some attributes: $x.sum$, that stores $\sum_{i \in x.interval} C[i]$; and $x.lazy$, that stores a value that need to be propagated to each descendant of x. This means that $x.sum$ might not be accurate at a given time for any of the requested operation.

Range operations. In both operations we traverse the tree recursively starting from the root and, at *each* recursive step on any internal node x, if $x.lazy \neq 0$: we set $x.sum \leftarrow x.sum + |x.interval| \times x.lazy$, we propagate the lazy information to x's children $x.left.lazy \leftarrow x.lazy$, $x.right.lazy \leftarrow x.lazy$ and, finally, we reset the information $x.lazy \leftarrow 0$. Afterwards, if the operation is

- $sum(i, j)$, we do the following:
  1. if $x.interval \cap [i, j] = \emptyset$, we return 0;
  2. if $x.interval \subseteq [i, j]$, we return $x.sum$;
  3. otherwise we repeat the procedure $sum(i, j)$ on $x.left$ and $x.right$, returning the sum of these calls.
- $update(i, j, c)$, we do the following:
  1. if $x.interval \cap [i, j] = \emptyset$, we stop the recursion on this subtree;
  2. if $x.interval \subseteq [i, j]$, we update $x.sum \leftarrow |x.interval| \times c$, and $x.left.lazy \leftarrow x.left.lazy + c$, $x.right.lazy \leftarrow x.right.lazy + c$
  3. otherwise we repeat the procedure $update(i, j, c)$ on $x.left$ and $x.right$.

The space occupied by T_I is $\sum_{i=0}^{\log_2 n} n/2^{-i} = 2n - 1$.

2 Depth of a node in a random search tree

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

1. the expected depth of a node (i.e. the random variable representing the distance of the node from the root) is nearly $2 \ln n$;
2. the expected size of its subtree is nearly $2 \ln n$ too, observing that it is a simple variation of the previous analysis;
3. the that the probability that the depth of a node exceeds $c2 \ln n$ is small for any given constant $c > 1$.

2.1 Proof with indicator variable

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

Proof. Let z_m the 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:

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

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

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

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

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

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

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

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

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

□

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

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

Prove that the that the probability that the depth of a node exceeds $c2 \ln n$ is small for any given constant $c > 1$.

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

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

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

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

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

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

□

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

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

2.2 Recursive balanced proof

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

- $height(T) = h \iff |T.left| = |T.right| \pm 1$ Trivially, let r be the root of a 3-nodes partition: then, if the partition is unbalanced, the lesser one will comprise of 0 nodes, while the greater one of 2, which implies that $height(T.right) == 2$.
- P = 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 2.2k-distance to a tree T with $n = 3$ nodes. Trivially, $height(T)$ with balanced tree is equal to two. Now, let us pick either the lowest or the greatest pivot possible: the tree is unbalanced towards either the left or the right, but $height(T) = 2$ in both cases. As the reader can see from 2.2, the distance works in absolute value; it is then clear how, at every permutation for a pivot p, out of the n, there are 2 that generate a tree of the same height: $p = d(P) + k, p = d(P) - k$. Given that at every iteration a node x in a completely unbalanced tree T' has a probability of $\frac{1}{n-i}$, we can define the probability of x being a pivot at level l as:

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

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

$$P(x_k) = \Sigma_{k=1}^{l-1}(\frac{1}{n-l+1}) \tag{6}$$

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

Upper bound. By hypothesis,

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

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

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

3 Karp-Rabin fingerprinting on strings

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

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

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

3.1 First solution

In order to save computational cycles on checks over ranges we use a similar structure to the one in the range updates: we compute the hashing on the first character in $O(1)$ time, then roll the hash through the $n-1$ remaining characters through $nO(1)$ operations. We call H this array; we also denote h_k as the function ca^i 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

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

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

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

```
1: function LCE(i, j, l):
2:     eq = 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}
```

3.2 Second solution

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

fingerprint on strings h_p as:

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

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

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

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

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

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

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

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

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

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

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

4 Hashing sets

Your company has a database $S \subseteq U$ of keys. For this database, it uses a hash function h uniformly chosen at random from a universal family $\mathcal{H}$ (as seen in class); it also keeps a bit vector B_S of m entries, initialized to zeroes, which are then set $B_S[h(k)] = 1$ for every $k \in S$ (note that collisions may happen). Unfortunately, the database S has been lost, thus only B_S and h are known, and the rest is no more accessible. Now, given $k \in U$, how can you establish if k was in S or not? What is the probability of error? [Note: you are not choosing k and S randomly as the they are both given... randomization here is in the choice of $h \in \mathcal{H}$ performed when building B_S.] Under the hypothesis that $m \geq c|S|$ for some $c > 1$, find the expected number of 1s in B_S under a uniform choice at random of $h \in \mathcal{H}$.

Solution. 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 $S \subseteq U$.

Let us now try and estimate $|cl_S(k)|$: given h is universal, we have an expected number of collisions of $\mathbb{E}[X_k] \approx \frac{1}{m} \forall k \in S$, that is

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

We can similarly compute the probability of not collision by simply replacing $\frac{1}{m}$ with $(1 - \frac{1}{m})$:

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

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

$$1 - \mathbb{P}\left[\text{S hit the given cell}\right] = 1 - \left(1 - \frac{1}{m}\right)^{|S|} \tag{8}$$

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

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

Therefore

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

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:

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

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

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

where $\mathbb{P}[B_s[k] = 1] = \frac{\alpha}{m}$ as α are the favourable 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 sufficiently large prime number $(m + 1 \leq p \leq 2m)$.

5.1 First proof

By linear algebra, $a + (kp) \bmod p = c \forall k \in \mathbb{N}$. For instance

- $(1 + 10 \cdot 0) \mod 10 = 1 \mod 10 = 1$
- $(1 + 10 \cdot 1) \mod 10 = 11 \mod 10 = 1$
- $(1 + 10 \cdot 2) \mod 10 = 21 \mod 10 = 1$
- ...

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 $k = \frac{p}{m}$ values for which $m_i \bmod p = d$ since $p > m$ and by 5.1. The same goes for x_1, x_2:

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

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):

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

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.

Since there are 2^b possible configurations of the memory, we choose to create a class of 2^b streams of length n. Each stream s has two occurrences of any symbol chosen from an alphabet Σ_s of size $n/2$. Therefore, every stream has tied frequencies for all its elements (until we generate the next one). The alphabets $\Sigma_1, \Sigma_2, \ldots \Sigma_{2^b}$ are all subset of Σ, that satisfies the following condition:

$$\binom{|\Sigma|}{n/2} > 2^b$$

With this condition we can generate 2^b streams (from just as many alphabets) such that, for any pair of streams, there exist at least one symbols that appear in one stream but not in the other.

We can apply the pigeonhole principle: suppose that the same configuration is reached in two different executions of the algorithm on two streams with symbols, respectively, from Σ_i and Σ_j. When we will emit as $(n+1)$-th character of the stream the symbol $a \in \Sigma_i$ such that $a \notin \Sigma_j$, 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. If $\mathcal{A} \neq a$, then the algorithm gave the wrong answer for the first stream. Otherwise, if $\mathcal{A} = a$, then it's correct for the first stream, but not for the second one, because $a \notin \Sigma_j$.

stream with symbols from Σ_i	...	d	d	e	e	f	f	g	g	h	h
stream with symbols from Σ_j	...	4	4	5	5	6	6	7	7	8	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.

Solution. We'll use the following notation:

- j^i is the i-th most frequent element.
- C is the frequencies function: $C(j^i) = \#\text{occurencies of } C^i$.

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.

Recursive sub-streams It is trivial to show that if we were to have $\frac{n}{2}$ counters,

$$\sum_{i=0, i\neq 1}^{\frac{n}{2}} \big(C(j^i) < C(j^1)\big)$$

by 7. We can also assert that given

$$S' = S[0, \frac{n}{2}], S'' = S[\frac{n}{2} + 1, n], k = \text{number of elements} \in S$$

- 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

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

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

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

```
1: function MAJORITY_VOTE_ALGORITHM(S):
2:     candidate ← null
3:     counter ← 0
4:     for all object ∈ S do
5:         if counter = 0 then
```

```
6:              candidate ← object                    ▷ candidate is no more dominant, swap it
7:          else if object = candidate then
8:              counter ← counter + 1
9:          else if object ≠ candidate then
10:             counter ← counter − 1
11:     return candidate
```

8 Count-min sketch: extension to negative counters

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

8.1 First solution

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

- a table T of size $r \times c$, where $r = \log \frac{1}{\delta}$ and $c = \frac{e}{\varepsilon}$, 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:

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

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

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

The probability that any count (estimate of $F[i]$) is off by more than $3\varepsilon\|F\|$ is:

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

The median is a bad choice if more than half of the $r = \ln 1/\delta$ estimates are bad. We define Y to be the number of bad estimates: $Y = \sum_{j=1}^{r} Y_j$, where Y_j is 1 if $|X_{i,j}| \geq 3\varepsilon\|F\|$ and 0 otherwise. Hence the median is a bad choice if the sum Y is at least $r/2$, and this happens with probability:

$$\mathbb{P}\left[Y > \frac{1}{2}\ln\frac{1}{\delta}\right]$$

We can apply the Theorem 1 (with $\mu = \mathbb{E}[Y] = \sum_{j=1}^{r}\mathbb{P}[j \text{ is a bad estimate}] < r/8$) to upper bound this probability. □

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 $\frac{r}{2}$

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

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 $\sum_{l=1}^{t} c_l > \sum_{k=i}^{j} F[k] + 2\varepsilon\ log\ n||F||$), 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 its length is a power of two ($l = 2^y$), and begins at a multiple of its own length: $[j2^y + 1, (j + 1)2^y]$. For example, $[13, 16]$ can be written as $[3 \cdot 2^2 + 1, (3 + 1) \cdot 2^2]$. Any arbitrary range of size s can be partitioned into $O(\log s)$ dyadic ranges, for example [2]:

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

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

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

The time to compute the estimate or to make an update is $O(\log n \log \frac{1}{\delta})$. The space used is $O(\frac{1}{\varepsilon} \log n \log \frac{1}{\delta})$, 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

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

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

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

Now, for the j-th estimation we can compute

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

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

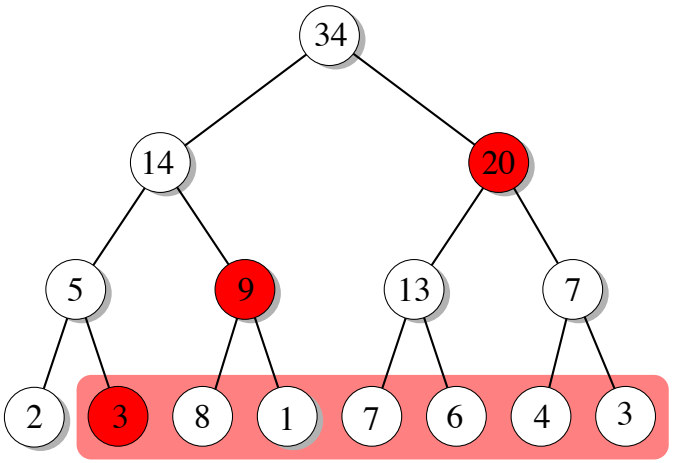


Figure 1: A hierarchy of dyadic ranges. The leaves are the ranges of length 2^0, while the root corresponds to the single range of length 2^3. Each level of the tree can be seen as a CM sketch table. To estimate $\sum_{k=2}^{8} F[k]$, the range $[2, 8]$ is decomposed into dyadic ranges $[2, 2], [3, 4],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

10 Space-efficient perfect hash

Consider the two-level perfect hash tables presented in [CLRS] and discussed in class. As already discussed, for a given set of n keys from the universe U, a random universal hash function $h : U \to [m]$ 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 bitvector, and $h_{h(k)}(k)$ is the offset for the secondary level.

Hash functions space optimization. We now try to improve the space for the hash functions parameters a_j, b_j, p_j. First, we select the lowest p_j possible, that is, the first $p_j > n_j^2$: by Bertrand postulate such a prime p_j exists for $n_j^2 \leq p_j \leq 2n_j^2 - 2$. Since (a_j, b_j) are chosen in $\mathbb{Z}_{p_j}^* \times \mathbb{Z}_{p_j}$ we need at most $\log_2(2n_j^2 - 2) < \log_2(2n_j^2) = 1 + 2\log_2 n_j$ bits for the binary representation of each parameter, and three times that space for the triple (a_j, b_j, p_j). Let us now compute the space for the n hash functions:

$$Y = \sum_{j=0}^{n-1} 3(1 + 2\log_2 n_j) = 3n + 6\sum_{j=0}^{n-1} \log_2 n_j \leq 3n + 6\sum_{j=0}^{n-1} n_j$$

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

$$2n + 3n + 3n + 12n = 20n \in O(n)$$

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.

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

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

$$C[rank_1(i) - 1] \stackrel{?}{=} s(l). \tag{16}$$

In fact, $rank_1(i) - 1$, executed on B, returns the position in C where we find the signature for the key $k \in S$ that caused $B[i] = 1$.

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

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

12 MinHash sketches

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

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

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 $1/\binom{n}{2}$, 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 $\geq 1/\binom{n}{2}$ [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 $1 - 1/n^c$ for a constant $c > 0$ (hence, with high probability).

Contraction step. 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 $Adj[uv]$. The total cost of this step is $O(n)$, as at most both lists containing n elements each are scanned.

Weighted graph extension. We now extend the above to weighted multi-graph by re-conducting it to the non-weighted case. First we'll define the *min-weighted cut*, the weighted cut where $\min_{|c|>0} C = \min_C \sum \omega(c)$, that is the sum of the weights determines the weight of the overall cut. We then have the weight for the overall graph:

$$\omega_G = \sum^{n} (\omega(n))$$

and for the min cut: ω_{min}. Similarly as we've seen in class we can define ω_G in function of ω_{min}:

$$\omega_G = \frac{\sum^{n} (\omega(n))}{2} = \frac{n \cdot \omega_{min}}{2}$$

Giving us an error probability of

$$\Pr(error) = \frac{\text{WEIGHT(MIN-CUT)}}{\text{WEIGHT(GRAPH)}} = \frac{\omega_{min}}{\omega_G} = \frac{\omega_{min}}{\frac{n \cdot \omega_{min}}{2}} = \frac{2}{n}$$

That is, we reach the same error probability seen in class. Therefore we can safely assume that the analysis is the same.

The algorithm does not need to be edited but in the computation of the min cut, which is now ω_c and not $\sum^{e^{i,j}} (e_i)$ where e_i is an edge between i, j last remaining nodes in the cut.

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

$$1 - \Pr(\text{success}) = 1 - \frac{1}{\binom{n}{2}}$$

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

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

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

14 External memory implicit searching

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

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

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

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

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

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

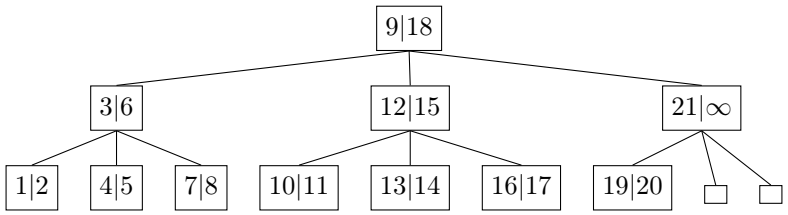
For example, suppose that $B = 2$ and the keys are $1, 2, \ldots, 21$. First, we select the keys to move to the upper level (those boxed), while the remaining keys will be the leaves of the tree:

$$1 \quad 2 \quad \boxed{3} \quad 4 \quad 5 \quad \boxed{6} \quad 7 \quad 8 \quad \boxed{9} \quad 10 \quad 11 \quad \boxed{12} \quad 13 \quad 14 \quad \boxed{15} \quad 16 \quad 17 \quad \boxed{18} \quad 19 \quad 20 \quad \boxed{21}$$

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

$$3 \quad 6 \quad \boxed{9} \quad 12 \quad 15 \quad \boxed{18} \quad 21$$

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



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

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

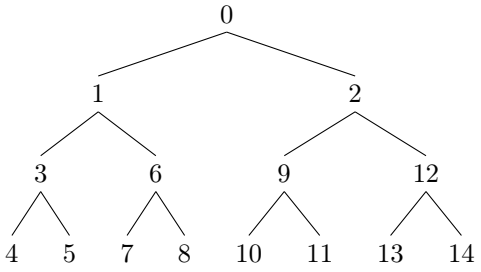
Space and I/O complexity. The construction algorithm copies the permutation of the N keys from A to a new file, and thus the total additional space needed in external memory is $O(N/B)$ blocks. After the preprocessing, A is discarded, hence the auxiliary space is $O(1)$.

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

15 Implicit navigation in vEB layout

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

Solution. We assume that the keys are stored in a zero-indexed array A s.t. $\log A = H$. Given and index $0 \leq i < A.length$ we will give a rule for computing the index of its children.



First, we define the *cut distance* for a given node v. Given that T has height $h = 2^k$, each cut halves the current sub-tree $T', height(T') = 2k'$ producing sub-trees of height $\frac{2^{k'}}{2} = 2^{k'-1}$. Since the base case for the construction algorithm is $height(T') = 2$, this operation is repeated until a sub-tree of height 2 is reached: therefore each node v is at distance $d \leq 2$ from its next cut in the construction.

Left and right children on $d(v) = 2$ By the above we have that on any even level (those for which $d(v) = 2$ holds) l will have a left child $v' = v + 1$ and a right child in position $v'' = v + 2$. As by construction the base case for $height(T') = 2$ is populated contiguously top-down left-to-right resulting in the above.

Left and right children on $d(v) = 1$ By the above we have that on any even level (those for which $d(v) = 1$ holds) l will have a left child $v' = v + \delta$ and a right child in position $v'' = v + \gamma$ for some δ, γ. The *vEB* construction allocates space for each index crossing a path in a top-down left-to-right, as stated previously. Now, since we are crossing a cut, we can either compute the size $|T_{ls}|$ of all the $2LS$ sub-trees allocated before the left child or the size of the tree T_{top} of height l on top of v and the size of the sub-trees on the left of our v.

$$|T_{top}| = 2^{l+1} - 1 \tag{17}$$

$$|T_{ls}| = 2^{H-l} \cdot 2LS \tag{18}$$

Therefore our left child as children given by the size of the top tree $T_{top}+$ the size of its left siblings sub-trees LSS:

$$|T_{ls}| + |T_{top}| = (2^{H-l})(2LS) + (2^{l+1} - 1) \tag{19}$$

A search algorithm is then straightforward: starting from the root we traverse the node as a binary tree, keeping track of the variables needed by 19, namely level and left siblings (each increasing by 1 and by a factor of 2 or $2 + 1$ respectively).

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

Require: S sorted in ascending order

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

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

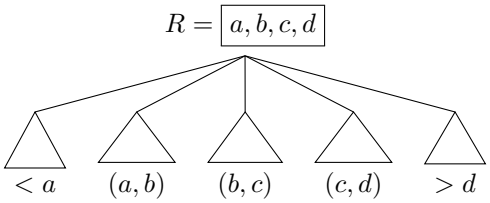
Heap tree in vEB layout. Given any array S sorted in increasing (decreasing) order, the implicit tree with root $S[i]$, children $S[\text{Left}(i)]$ and $S[\text{Right}(i)]$, satisfies the min-heap (max-heap) property $\forall i \in [0, S.length - 1]$.

16 1-D range query

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

16.1 First solution

B-tree. To report the keys in the given range we perform an inorder traversal of the tree starting from the root R: for each key k stored in the node R in position i, if $k \geq k_1$, then we traverse the i-th subtree of R and output k; if $k > k_2$, then we traverse the i-th subtree of R and stop the for loop.



First, this algorithm searches for the key k_1 and, in the worst case, it visits all the nodes in a walk from the root to a leaf with $O(\log_B N)$ block transfers (see Figure 2). Afterward, it starts to report all the keys in the given range: this can be done with $O(R/B)$ block transfers, provided that we can hold in memory one block for each level of the tree, which is stored contiguously in memory and scanned left-to-right. Consequently, the I/O complexity of the range query is $O(\log_B N + R/B)$.

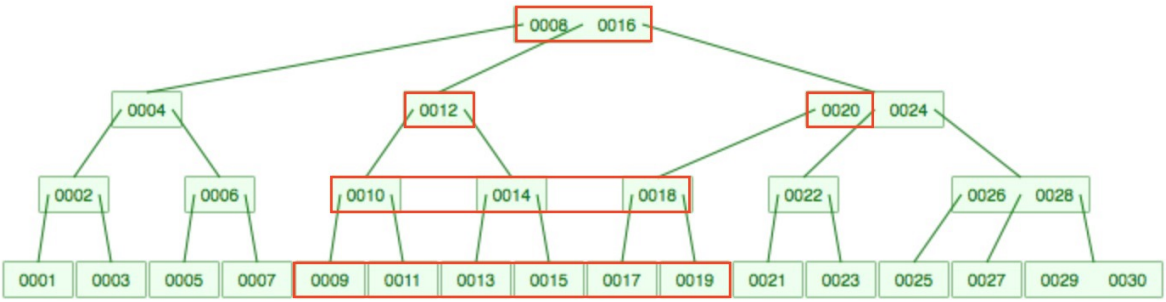


Figure 2: A B-tree with the keys in range $[8, 20]$ highlighted in red.

16.2 Second solution

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

at most $h'' + \frac{R''}{B}$. Since we loaded only blocks whose keys belong to $[k_1, k_2]$ we have $R' + R'' = R$ and $h'' + \frac{R''}{B} + \log_B h' + \frac{R'}{B} = 2\log_B N + \frac{R}{B}$ giving an output-sensitive search algorithm.

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

17 External memory mergesort

In the external-memory model (hereafter EM model), show how to implement the k-way merge (where $(k+1)B \leq M$), namely, how to simultaneously merge k sorted sequences of total length N, with an I/O cost of $O(N/B)$ where B is the block transfer size. Also, try to minimize and analyze the CPU time cost.

17.1 First solution

We keep in main memory k input buffers $B_1, B_2, \ldots, B_k$ — one for each sorted sequence $s_1, s_2, ..., s_k$ that we need to merge — and 1 output buffer B_{out}. At each step we find the smallest unchosen element among the input buffers and copy this element to B_{out}. When B_{out} is full, we write it to the end of a file F, then we clear B_{out}'s content.

For each input buffer we store a pointer p_i to its first unchosen element. When a pointer p_i reaches the end of the buffer B_i, we need to load in B_i the next portion of the corresponding sorted sequence s_i and reset p_i. We also store a pointer to the next free slot in B_{out}.

I/O complexity. The k-way merge ends when we reach the end of every sorted sequence: we performed $\Theta(N/B)$ read operations for the sorted sequences, and $\Theta(N/B)$ writes to F, thus the total I/O complexity is $\Theta(N/B)$.

Minimizing CPU time. The CPU complexity of searching the smallest element among the unchosen elements of the input buffers is $O(k)$. We can improve the algorithm by replacing the linear search with a minimum priority queue of the unchosen elements: in this way we can extract the smallest element in $O(1)$, advance the pointer p_i of the corresponding input buffer B_i and finally insert $B_i[p_i]$ in the priority queue in $O(\log k)$ time.

17.2 Second solution

Let

- $k = \frac{1}{4} \cdot \frac{M}{B}$ be the number of blocks the cache can hold.
- B_s the blocks reserved in cache to store temporary merges.
- N be the size of the problem.
- B the block size, with B_i a pointer to the i^{th} block.
- B_i^c the i^{th} block stored in memory.
- $\frac{N}{B}$ the number of blocks storing A in main memory.

We assume the sorted blocks we receive in input are ordered and use $\frac{1}{3} \cdot \frac{M}{B}$ to store loaded blocks in a Fibonacci min-heap fashion. *k-way* mergesort operates on k sorted blocks we load in cache. We implement the merge operation as follows, outputting to external memory on file O:

```
1:  function MERGE(a, b):
2:      for i ← 0; i < k; i++ do
3:          load(B_i, B_i^m)                          ▷ Load blocks in cache: O(N/B)
4:      busy_blocks = k − 1                           ▷ Count busy blocks, that is blocks whose pointers
                                                        have not reached the end of the block
5:      S ← &S                                        ▷ At most O(k·B), when every sorted block is greater
                                                        than the following
6:      while busy_blocks > 0 do
7:          m ← min{min B_i^c}                        ▷ Get the current minimum of every block in cache: O(k)
8:          B_j^c ← B_j^c + 1                         ▷ Shift to next element in block B_j with current minimum element
9:          if B_j == EOB then
10:             busy_blocks ← busy_blocks − 1
11:         if busy_blocks == 1 then
```

12: $S \leftarrow S :: flush(B_j^c)$ ▷ Flush remaining block

13: $*S \leftarrow m$

14: $S \leftarrow S + 1$ ▷ Increment storing blocks pointer

15: write(S, O) ▷ Output

As we assumed, our blocks are ordered Fibonacci min-heaps, thus allowing us to merge them in $k \cdot \log(n)$ time. We can then store S by finding the minimum element n times in costant time: $n \cdot O(1) = O(n)$, giving us a total cost of

$$k \cdot B + \log(k) \cdot O(k) + O(n)$$

respectively for construction, find the minimum over k Fibonacchi heaps in the *merge* procedure and store it in memory.

18 External memory (EM) permuting

Given two input arrays A and π^{-1}, where A contains N elements and π^{-1} contains a permutation of $\{1, \ldots, N\}$, describe and analyse an optimal external-memory algorithm for producing an output array C of N elements such that $C[i] = A[\pi[i]]$ for $1 \leq i \leq N$.

Solution.

1. We define an array π^{-1} such that $I\pi^{-1}[i] = (\pi, i)$ of the form (departure, destination): note as we need $O(\frac{N}{B})$ block transfers.
2. We then sort π^{-1} according to the departure with the previously defined k-way mergesort in $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$ block transfers. π^{-1} now holds the ordered departures with respective index where to send the elements to (the destination).
3. We can now build $A\pi^{-1}$ with entries $(\pi^{-1}[i].destination, A[i])$ of the form (destination, element). As we ordered by the departures we scan A sequentially: $A\pi^{-1} = [(destination, A[0]),$ $(destination, A[1]), (destination, A[2]), \ldots]$: again we have $O(\frac{N}{B})$ block transfers.
4. We run one more ordering over such tuples in $A\pi^{-1}$, this time according to the destination: we now have an ordered mapping $destination(\pi[i]) \rightarrow element$ and we are able to write it to memory with $O(\frac{N}{B})$ block transfers:

$$sort(A\pi^1) = [(0, A[x]), (1, A[x]), (2, A[y]), (3, A[x]), \ldots] =$$
$$= [(0, A[\pi[0]]), (1, A[\pi[1]]), (2, A[\pi[2]]), (3, A[\pi[3]]), \ldots]$$

The I/O complexity of the solution is dominated by the cost of sorting, that is $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$.

19 Suffix sorting in EM

Using the DC3 algorithm seen in class, and based on a variation of mergesort, design an EM algorithm to build the suffix array for a text of N symbols. The I/O complexity should be the same as that of standard sorting, namely, $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$ block transfers.

Solution . We improve the *DC3* algorithm by exploiting the multi-way mergesort defined in exercise 17. To give a proper overview we'll insert the complete algorithm, highlighting the differences introduced.

Samples construction We split our string A of size $3N$ in two chunks S_0, S_1, S_2 each of size N according to the index of each character: $c \in S_i \iff i \mod 3 = i$. We are able to do so in linear $O(N)$ time and $\frac{N}{B}$ cache loads, as a linear scan is sufficient.

Sample sorting We now twitch a little bit the *DC3*: instead of operating a traditional radix-sort in order to sort $S_{1,2}$ we apply our multi-way merge-sort on $k = \frac{M}{B}$ *3-grams*, ordering them in $\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B}$. The use of multi-way merge-sort allows us to reduce computation time and cache cost, thus allowing us to stay in the boundary of $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$. Storage of $S^-1[A]$ still takes linear time and $\frac{M}{B} + \frac{1}{3} \cdot \frac{M}{B}$ cache writes, as $\frac{1}{3} \cdot \frac{M}{B}$ single values (the ranks) need to be stored.

Non-sample sorting As operations over S_0 are trivial lexicographic comparisons, we are able to load them in $\frac{3N}{B}$ cache loads and sort them using again our multi-way merge-sort obtaining the above results. We can further improve by then writing the merge 3-tuples of $S_{0,1}, S_{0,2}, S_{1,2}$ to disk in linear time: as they are constructed with $\frac{M}{B}$ cache loads for $S_{1,2}$, $\frac{M}{B}$ cache loads for S_0 and $\frac{M}{B}$ cache loads for their respective ranks.

Merging We then merge the ranks computed in the sample and non-sample sorting of step 2 and 3 of the algorithm with the multi-way merge-sort whose comparison operator is the same used by the classical *DC3* algorithm.

20 Wrong greedy for minimum vertex cover

Find an example of (family of) graphs for which the following greedy approach fails to give a 2-approximation for the minimum vertex cover problem (and prove why this is so). Start out with an empty $\tilde{S}$. Choose each time a vertex v with the largest number of incident edges in the current graph. Add v to $\tilde{S}$ and remove its incident edges. Repeat the process on the resulting graph as long as there are edges in it. Return $|\tilde{S}|$ as the a approximation of the minimal size of a vertex cover for the original input graph. Generalize your argument to show that the above greedy algorithm cannot actually provide an r-approximation for any given constant $r > 1$.

Solution We create a class of graphs $G = (V, E), |V| = N$ s.t. V is partitioned in S, R s.t. $|S| = k, |R| = \sum_{i=1}^{k} \left(\lfloor \frac{k}{i} \rfloor\right)$ respectively and define them as *senders* and *receivers*.

We then build the edges in the following way in order to obtain a minimum cover L:

1. For the current iteration i, if $\lfloor \frac{k}{i} \rfloor = 0$ we stop.
2. Split the current slice R, consider the first $\lfloor \frac{k}{i} \rfloor$ vertexes: let R^i be this slice.
3. For every vertex $v \in R^i$ build $\frac{m}{\lfloor \frac{k}{i} \rfloor}$ outgoing edges to each of the m nodes $\in R$. We can then repeat step 1 with $R \leftarrow R \setminus R^i$.

Once the above procedure ends, we have partitions vertexes of size $k \in S : deg(v) = 1$, one partition of vertexes of size $\lfloor \frac{k}{2} \rfloor \in S : deg(v) = \lfloor \frac{k}{2} \rfloor \forall v \in S$, etc. Each node in R will have an incoming edge for every node $n \in L$ and will therefore be in the minimum vertex cover, as by hypothesis $|S| > |R|$ The greedy algorithm then starts by removing the highest-degree node: we find this on top of R, as it was doubled at each iteration. By construction we have k *senders* partitions whose highest degree is k (note as the degree increases in the summation up to $\lfloor \frac{k}{i} \rfloor$ for $i = k$). The *receivers* have a maximum degree of $\sum_{i=1}^{k} \lfloor \frac{k}{i} \rfloor \approx k \ln k$, as each of them received an incoming edge from each partitions $\in S$ that we asserted being $\lfloor \frac{k}{i} \rfloor$ by construction. The greedy algorithm will start removing from the vertex with highest degree: it will be forced to remove from the last built slices up to $\lfloor \frac{k}{i} \rfloor > \frac{m}{\lfloor \frac{k}{i} \rfloor}$, thus cutting at least $k \ln k > m$ vertexes before cutting in the minimum cover S. We can build a family of such graphs by making k grow at will: as $k \ln k > r \forall k$ such family will have an increasingly larger *r-approximation*:

$$k \ln k > rk \ \forall k, r$$

21 Greedy 2-approximation for MAX-CUT on weighted graphs

Prove that the greedy algorithm for *MAX-CUT* described in class gives also a 2-approximation for weighted graphs with positive weights.

Solution. Let G be our undirected weighted graph with n nodes and m edges. As stated in class, the greedy algorithm scans sequentially the ordered nodes, deciding at each iteration i whether to add the node v to the cut or not by computing the following:

1: **function** ADD?(v, S)
2: $S' \leftarrow S \cup \{v\}$
3: **if** $E(S') > E(S)$ **then**
4: $S \leftarrow S'$

With a weighted graph we need to slightly modify the above:

1: **function** ADD?(v, S)
2: $S' \leftarrow S \cup \{v\}$
3: **if** $\sum_{n \in S'} w(n) > \sum_{n \in S} w(n)$ **then**
4: $S \leftarrow S'$

The above is computed for each vertex v in local search but only $r_v = |E_v|, E_v = \{v' | \exists e(v + \delta, v) \in E\}$ for the greedy algorithm, that is the nodes with a greater ordering and with a colliding edge on v. By applying the above function we have an r'_v. We also know that in a non-weighted case, given any vertex u, at least $\frac{r_u}{2}$ vertexes are in the cut: if this was not true, then we could add one cut, switching v from S to $\bar{S}$ or vice versa increasing the number of nodes in the cut. A weighted variant is trivial: as the non-weighted case chooses to switch v according to $\sum r^i$, we choose to switch according to the weight sum $\mathcal{W} = \sum \omega(r^i)$. It follows that $r'_v = \sum \omega(\text{edges incident to v in the cut}) \geq \sum \frac{\omega(\text{incident edges on v})}{2}$. Therefore we have the weights in the cut

$$E_\omega = \sum_{v \in V} \frac{\omega(r'_v)}{2} = \frac{1}{2} \sum_{v \in V} \omega(r'_v) = \frac{1}{2} \mathcal{W}(\mathcal{G}) \tag{20}$$

22 Randomized 2-approximation for MAX-CUT

Prove that the following randomized algorithm provides a 2-approximation for MAX-CUT in expectation, namely, the expected cut size is at least half of the optimal cut size. Here are the steps. *(1)* For each vertex $v \in V$, toss an unbiased coin: if it is tail, insert v into C; else, insert v into $V \setminus C$. *(2)* Start out with an empty set T. For each edge $\{v, w\} \in E$, such that $v \in C$ and $w \in V \setminus C$, add $\{v, w\}$ to T. Return $|T|$ as approximated solution.

Solution. We define the indicator variable $X_{v,w}$:

$$X_{v,w} = \begin{cases} 1 & \text{if } \{v, w\} \in T \\ 0 & \text{otherwise} \end{cases}$$

with $\Pr(X_{v,w} = 1) = \Pr(v \in C, w \notin C || v \notin C, w \in C) = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$ and expected value $\mathbb{E}[X] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 0 = \frac{1}{2}$. We then define the indicator variable X for all the cuts we might have:

$$X = \sum_{(v,w) \in E} X_{v,w}$$

with it expected value $\mathbb{E}[X]$:

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{(v,w) \in E} X_{v,w}\right] = \sum_{(v,w) \in E} \mathbb{E}[X_{v,w}] = \frac{1}{2}|E|$$

Given $|OPT| \leq |E|$ optimal cut and X, the cut we obtain through the above algorithm:

$$\mathbb{E}[X] = \frac{1}{2}|E| \leq \frac{1}{2}|OPT|$$

23 Approximation for MAX-SAT

In the MAX-SAT problem, we want to maximize the number of satisfied clauses in a CNF Boolean formula. Consider the following approximation algorithm for the problem. Let F be the given formula, $x_1, x_2, \ldots, x_n$ its Boolean variables, and $c_1, c_2, \ldots, c_m$ its clauses. Pick arbitrary Boolean values $b_1, b_2, \ldots, b_n$, where $b_i \in \{0, 1\}$ $(1 \leq i \leq n)$. Compute the number m_0 of satisfied clauses by the assignment having $x_i := b_i$ $(1 \leq i \leq n)$. Compute the number m_1 of satisfied clauses by the complement of the assignment, namely, having $x_i := \overline{b}_i$ $(1 \leq i \leq n)$, where $\overline{b}_i$ denotes the negation (complement) of b_i. If $m_0 > m_1$, return the assignment $x_i := b$ $(1 \leq i \leq n)$; else, return the assignment $x_i := \overline{b}_i$ $(1 \leq i \leq n)$. Show that the above algorithm provides an r-approximation for MAX-SAT, and specify for which value of $r > 1$ (explaining why). Discuss how the choice of $b_1, b_2, \ldots, b_n$ can impact the value of r, giving an explanation in your discussion. Optional: create an instance of the MAX-SAT problem where the returned value is exactly $1/r$ of the optimal solution, specifying which values of $b_1, b_2, \ldots, b_n$ have been employed.

Solution. Let $b = (b_1, b_2, \ldots, b_n)$ the assignment and $\overline{b}$ its bitwise complement. Any clause in F is always satisfied by b or $\overline{b}$ (or both). Let u be the number of clauses only satisfied by b, v the number of clauses only satisfied by $\overline{b}$, and w the number of clauses satisfied by both b and $\overline{b}$.

For example, in $F = c_1 \wedge c_2 \wedge c_3 = (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_2) \wedge (\neg x_3 \vee x_4)$, the assignment $b = (1, 0, 1, 1)$ satisfies c_1, c_2, c_3, while $\overline{b} = (0, 1, 0, 0)$ satisfies c_2, c_3, thus $u = 1, v = 0, w = 2$.

We can rewrite and bound the approximation $\widetilde{C} = \max\{m_0, m_1\}$ returned by the algorithm as

$$\widetilde{C} = \max\{u + w, v + w\} = w + \max\{u, v\} \geq w + \frac{u + v}{2} \geq \frac{1}{2}(u + v + w).$$

Since the optimal solution C^* is at most the number of clauses m, which in turn is at most $u + v + w$, it follows that

$$\frac{C^*}{\widetilde{C}} \leq \frac{u + v + w}{\frac{1}{2}(u + v + w)} = 2.$$

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]).

Solution. We should note that the following properties hold:

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

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

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

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

[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.

[4] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.