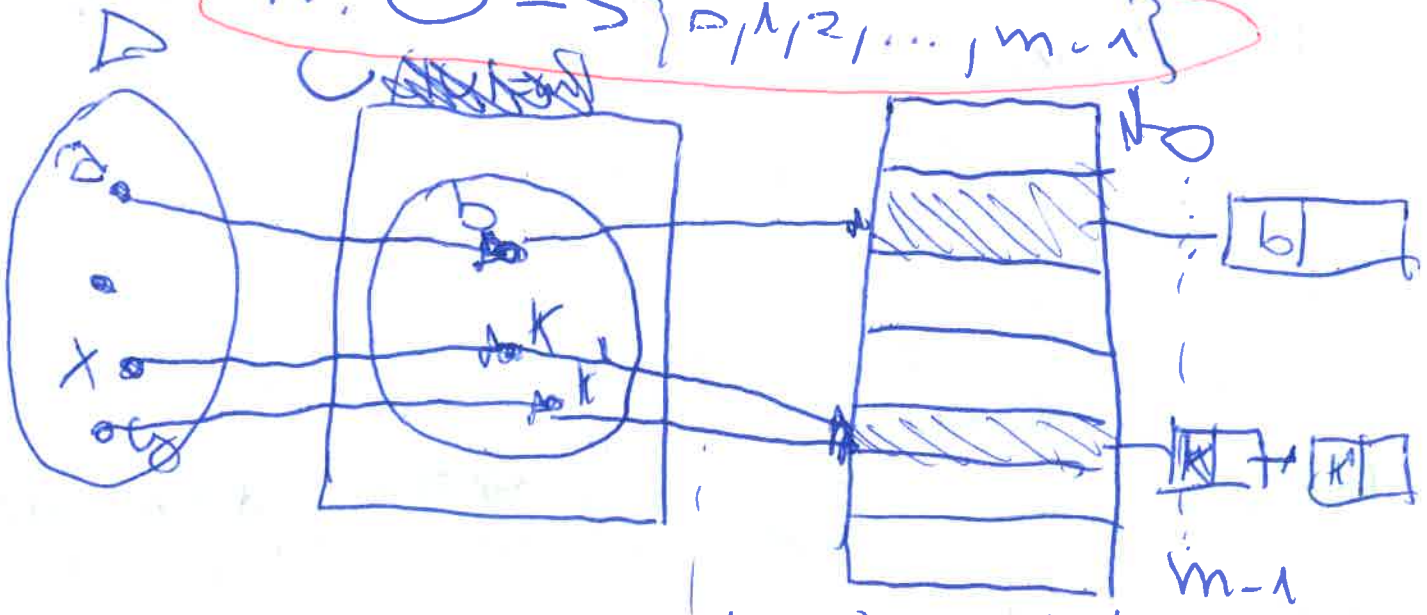
HASHING – HASH TABLES

HASHING WITH CHAINING:

The hash table is an array T of size m, whose entries are either NULL or they point to lists of dictionary items.

The mapping of items to array entries is implemented via a HASH FUNCTION.

$h : U \to \{0, 1, 2, \ldots, m-1\}$



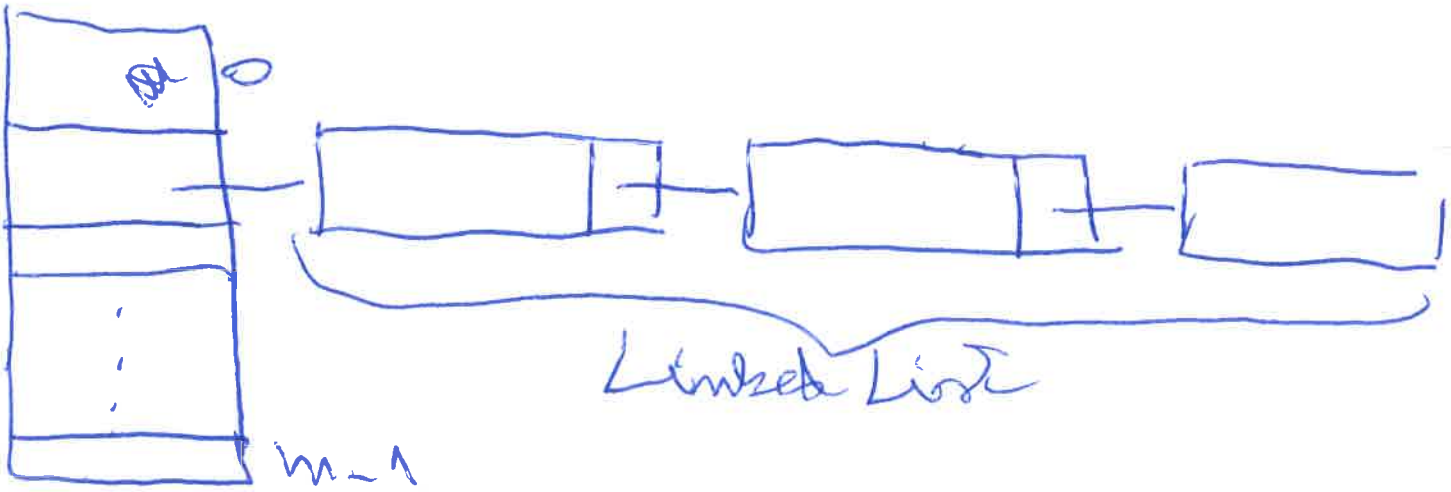
- $h(k) = h(k')$ [some hashing]

PROPERTIES OF HASH FUNCTION h:

- h is a SIMPLE, UNIFORM hash,

[i.e.: it can map any value to any possible cell with the same probability]

UNIFORM PROBABILITY.



$E\{\text{list length}\} = \frac{n}{m} = \frac{\#\text{keys}}{\text{TABLE SIZE}} = \alpha$ = LOAD FACTOR (average # of keys hashing to same value)

Generally, we have used hash functions of the following form:

$$h(k) = k \bmod p$$

Do such functions always satisfy the UNIFORM HASH property? No!

FAMILY of SIMPLE, UNIFORM HASH FUNCTIONS

RECOMMENDS for a HASH FUNCTION:

- FAMILY of HASH FUNCTIONS.

$h \in \mathcal{H}$: $m \cdot m \cdot \ldots \cdot m = m^U$ (k times; m different hash functions)

↳ Families of SIMPLE, UNIFORM HASH FUNCTIONS

↳ In a family of hash functions a hash function can map arbitrarily all keys to all possible values.

SPACE for the REPRESENTATION (in BITS)

$\forall h \in \mathcal{H}$, we need $\geq \log_2 m^U$ BITS,

$= U \cdot \log_2 m$ BITS

↳ IMPRACTICAL REPRESENTATION & even single HASH FUNCTION.

UNIVERSAL HASHING:

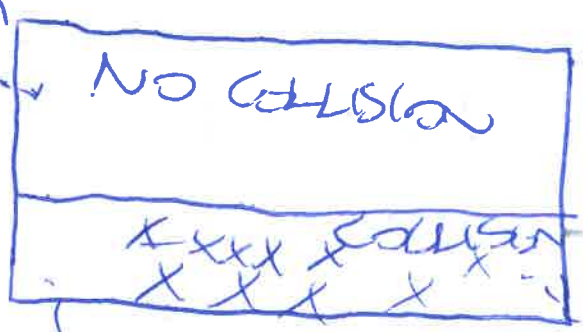
A family $\mathcal{H}$ of hash function is an UNIVERSAL HASHING iff:

$$\forall x, y \in U : \left[\#h : h(x) = h(y)\right] \leq \frac{|\mathcal{H}|}{m}$$

(COLLISION)

VISUALIZED SITUATION (i.e. P of collision occurring $\geq \frac{1}{m}$):

$h(x) \neq h(y)$:



$h : U \to \{0, \ldots, m-1\}$

$h(x) = h(y)$

$\to h(x) = h(y)$

HALF of all colliding functions

$\frac{|\mathcal{H}|}{m}$

ASSUME that one such family $\mathcal{H}$ exists.

$\Rightarrow$ Use that to pick a hash function (at random) that would make keys collide?

[i.e.: we would end up in the "collision part"]

~~Pr{h ∈ ℋ : h(x) = h(y)}~~

$\Pr_{h \in \mathcal{H}} = \Pr\{h(x)=h(y)\}$

$= \dfrac{|\mathcal{H}|/m}{|\mathcal{H}|}$ } colliding hash functions / } All possible hash functions $= \dfrac{1}{m} = \Pr\{$ hash function picked at random makes x, y collide $\}$

DEFINITION of ~~the~~ UNIVERSAL HASHING:

~~[crossed out]~~ In a UNIVERSAL HASHING FAMILY, a randomly-chosen hash function (h) from the set ℋ has P. to make distinct keys x and y to collide no more than ($\leq$) $\dfrac{1}{m}$.

LENGTH of CHAINING LISTS in UNIVERSAL HASHING:

"Let $T[0, m-1]$ be a hash table with chaining, where the hash function is picked randomly from a universal class ℋ.

$\Rightarrow$ The expected length of the chaining lists is still no more than $1+\alpha$, where $\alpha = \dfrac{n}{m}$

[i.e. a hash function taken from the family ℋ still has load factor $\alpha = \dfrac{n}{m}$!]

PROOF:

⇒ The expected load factor depends on the choice of h from $\mathcal{H}$, where $\mathcal{H}$ is a FAMILY of UNIVERSAL HASH FUNCTIONS

⇒ We hence define:

$$I_{x,y} = \begin{cases} 1 & \text{if } h(x) = h(y) \text{ (keys collide)} \\ 0 & \text{otherwise} \end{cases}$$

$\forall x, y \in S$

↳ INDICATOR R.V

Where $I_{x,y}$ is an INDICATOR R.V.

~~We want to determine the # keys where we have a collision, i.e.~~

~~$\# y \in D \mid h(x) = h(y)$~~

We are interested in determining the length of the chain for finding the LOAD FACTOR.

i.e.: $\sum_{y \in D} I_{x,y}$ = Length of chain.

where $\# y \in D : h(x) = h(y)$

$$E\left\{\sum_{y \in D} I_{x,y}\right\} = \sum_{y \in D} E\{I_{x,y}\}$$

$$E\{I_{x,y}\} = P\{I_{x,y} = 1\} * 1 + P\{I_{x,y} = 0\} \cdot 0$$

$$\Rightarrow E\{I_{x,y}\} = P\{I_{x,y}\} \cdot 1$$

$$\Rightarrow \sum_{y \in D} E\{I_{x,y}\} = \sum_{y \in D} P\{I_{x,y}=1\}$$

$$= \sum_{\substack{y \in D \\ (y \neq x)}} P\{h(x)=h(y)\} = \frac{n-1}{m} \leq \frac{n}{m} = \alpha$$

(because of exclusion of x)

$\Rightarrow$ We have shown that we STILL have the classic hashing behaviour. (on AVG.)

TIME COMPLEXITY – HASH FUNCTION FROM UNIVERSAL FUNCTIONS' FAMILY

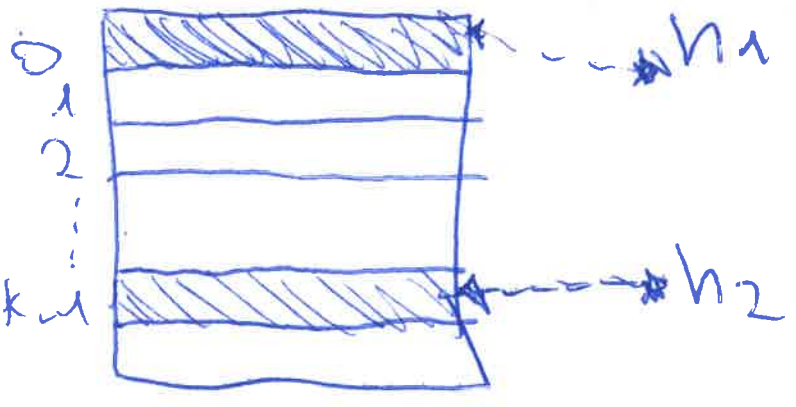
We known: • AVG. length of a LIST = $O(\frac{n}{m}) = \alpha$

WORST-CASE

• MAX. length of a LIST = $O(\frac{\log n}{\log \log n})$ (with high probability)

↳ Even if not constant time, we still have very little time.

* The POWER of 2 CHOICES:



MARVELOUS IDEA: Compute hash value by 2 hash functions, then ~~apply the~~ insert

the hashed value into the chain that has the least LOAD FACTOR $\alpha = \frac{n}{m}$

SPACE $= O(\log \log n) = (m)$

(previously $\log n \rightarrow \log \log n$)
reduced by one order of magnitude.

TIME $= O\left(\frac{\log n}{\log \log n}\right)$

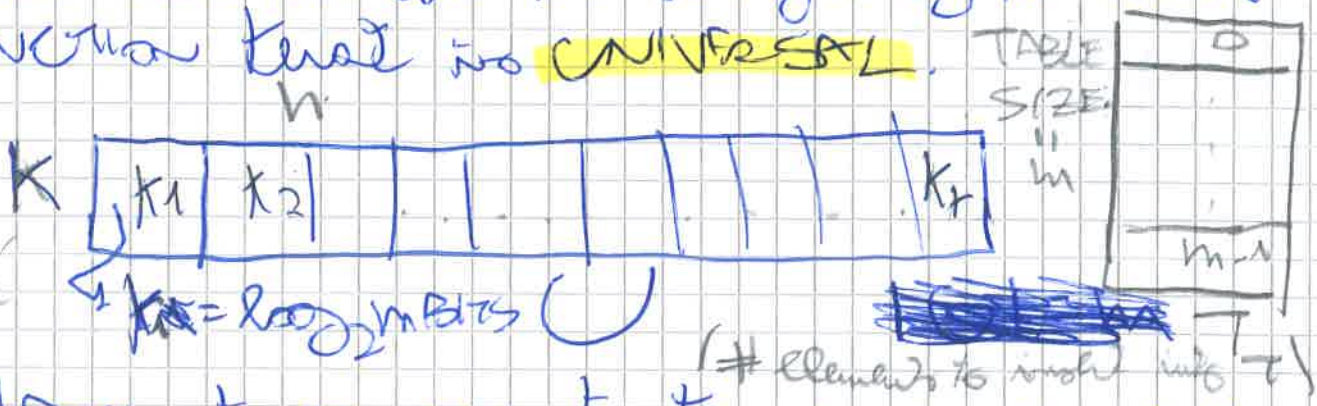
NO PROOFS!

NB: Operating Systems use this kind of hash to reduce space occupancy and Standard hash tables.

(NO PROOF!)

UNIVERSAL HASH FUNCTIONS | DESIGN:

We are interested in designing a HASH FUNCTION that is UNIVERSAL.



BITS to represent the HASH UNIVERSE $= \log_2 U$, where $r = \frac{\log_2 U}{\log_2 m}$ (log decomposes in base $|K|$)

UNIVERSAL FAMILY of HASH FUNCTIONS (with parameter 2) ↳ defines HASH FUNCTION!

A family of hash functions with parameter (2) is defined as:

$$\mathcal{H} = \{ h_2 : U \to \{0, 1, \ldots, m-1\} \}$$

(i.e.: for every integer parameter (2), you have the following UNIVERSAL hash function):

$$h_2(k) = \sum_{i=1}^{r} a_i k_i \mod m \Rightarrow \text{TABLE SIZE}$$

↳ UNIVERSAL HASH FUNCTION ↳ PRIME!

SITUATION VISUALIZED

⇒ Key (K) is split into (r) parts of $\log_2 m$ bits each

UNIVERSAL HASH FUNCTION'S DEFINITION

h_2 is a UNIVERSAL HASH FUNCTION.

$$\{\#h_2 : h_2(x) = h_2(y)\} \leq \frac{|\mathcal{H}|}{m}$$

(The #HASH FUNCTIONS where we have a collision is $\leq \frac{|\mathcal{H}|}{m}$ (for a random $h \in \mathcal{H}$ chosen from $\mathcal{H}$, the $P\{h(x) = h(y)\} \leq \frac{1}{m}$))

PROOF of the EXISTENCE of UNIVERSAL HASH FUNCTIONS

We want to show that the INEQUALITY is indeed satisfied!

$\Rightarrow$ TWO DISTINCT KEYS! $m \rightarrow$ PRIME NUMBERS.

For $x_0 \neq y_0$ we adopt a COUNTING APPROACH (count the #HASH FUNCTIONS where this occurs)

$$h_2(x) = h_2(y) \leftrightarrow \sum_{i=0}^{r-1} a_i x_i \equiv \sum_{i=0}^{r-1} a_i y_i \bmod m$$

$$\leftrightarrow a_0 x_0 + \sum_{i=1}^{r-1} a_i x_i \equiv a_0 y_0 + \sum_{i=1}^{r-1} a_i y_i \bmod m$$

$$\leftrightarrow a_0 \underbrace{(x_0 - y_0)}_{\neq 0} \equiv \sum_{i=1}^{r-1} a_i (y_i - x_i) \bmod m$$

[Because $x_0 \neq 0$ and $y_0 \neq 0$]

$\Rightarrow x_0 - y_0 \neq 0$.

$$a_0 \equiv \frac{1}{(x_0 - y_0)} \sum_{i=1}^{r-1} a_i (y_i - x_i) \bmod m$$

⇒ The last equation obtained for a_0 shows that:

↳ whatever is the choice for $[a_1, a_2, \ldots, a_{r-1}]$

there exists ONLY one choice for a_0 that causes x and y to COLLIDE.

(i.e. $h_a(x) = h_a(y)$).

$[a_1, \ldots, a_{r-1}]$

⇒ What is the #CHOICES for $[a_1, a_2, \ldots, a_{r-1}]$ that cause x and y to collide?

$[a_1, a_2, \ldots, a_{r-1}]$ ⇒ We have $r-1$ choices of m!

$m \quad m \quad \ldots \quad m$ ($r-1$ times!)

#CHOICES for a that cause x and y to collide $= m^{r-1} = \frac{m^r}{m} = \frac{U}{m} = \frac{|\mathcal{H}|}{m}$

⇒ $\{\# h_a : h_a(x) = h_a(y)\} \leq \frac{|\mathcal{H}|}{m}$ Q.E.D

(As we have as many hash functions h_a as parameters a)

↳ i.e. the definition at the beginning holds!

TWO-UNIVERSAL HASH FUNCTIONS:

We now check whether there exist other ways of constructing hash function, each as parameter a.

Consider $|U| = 2^h$ and $m = 2^l \leq |U|$

(l: a power of 2)

a = odd integer $< 2^h$

$$h_a(k) = ((ak \bmod 2^h) \text{ DIV } 2^{h-l})$$

↳ TWO-UNIVERSAL HASH FUNCTION.

Where $2h \geq 2^h$

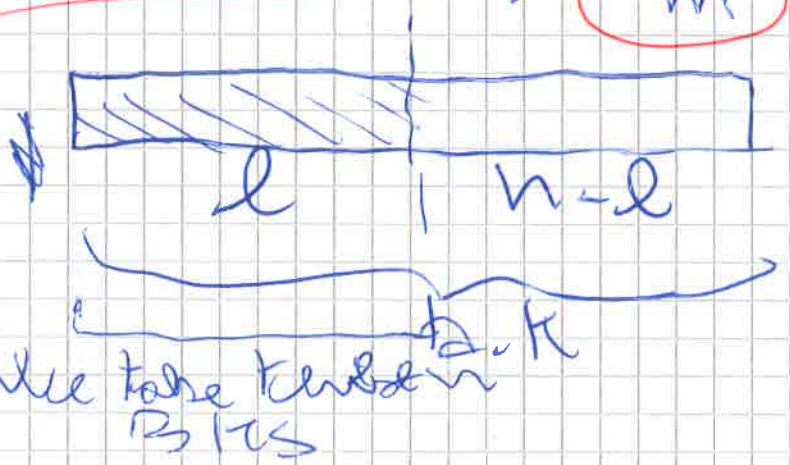
No PROOF of correctness, though we have:

$$P\{h_a(x) = h_a(y)\} \leq \frac{1}{2^{l-1}} = \frac{2}{m}$$

NB. We can also shuffle bags & use a mask to take the relevant BITS, or draw ⇒

P_r[a collision occurs for a UNIVERSAL HASH FUNCTION] $= \frac{1}{m}$

P_r[a collision occurs for a TWO-UNIVERSAL HASH FUNCTION] $= \frac{2}{m}$



PERFECT HASHING, MINIMAL ORDERED.

So far, hashing with chaining allows to search in $\Theta(\alpha)$ time on AVERAGE, where $\alpha = \Theta(\frac{n}{m})$ if $n \approx m$.
(i.e. constant time on AVERAGE!)
In the worst case, we would have LINEAR TIME access

⇒ FEATURES of PERFECT HASH FUNCTION: We want to achieve $O(1)$ in the SEARCH TIME access, with NO waste of space ($m = n$) still!
(OPTIMAL linear space + CONSTANT search time)

PERFECT HASH FUNCTION - DEFINITION:
A hash function is said to be perfect with respect to a dictionary D if:

$\forall x, y \in D: h(x) \neq h(y)$ (no collisions) $\Leftrightarrow \frac{|D|}{m} \leq 1$ (at most one element per slot)

2. ORDER-PRESERVING HASH FUNCTION.

A (minimal) perfect hash function is said to be ORDER-PRESERVING if: $k_i < k_j$

$\forall k_i, k_j \in S: h(k_i) < h(k_j)$

$\Rightarrow h(k)$ return RANK of k in the ordered dictionary S. [Only in STATIC SCENARIO case]

→ MINIMAL HASH FUNCTION.

A hash function is minimal if its hash table size is $m = n$.

[Smaller possible size of HASH TABLE].

LEMMA To define a HASH TABLE.

If S consists of q keys and $m = q^2$

then a CANTELLI HASH FUNCTION

DESIGN of a SIMPLE & PERFECT HASH TABLE:

If ordering and minimality is NOT required, then the design of a (static) perfect hash function is simpler.

→ Use a TWO-LEVEL HASHING SCHEME with universal hashing functions at each level.

TWO-LEVEL HASHING SCHEME

To design a SIMPLE & PERFECT hash table, we make use of 2 LEVELS of HASHING:

- **FIRST-LEVEL HASHING**: Analogously to hashing with chaining:

  keys are HASHED into m SLOTS via a UNIVERSAL HASH FUNCTION h

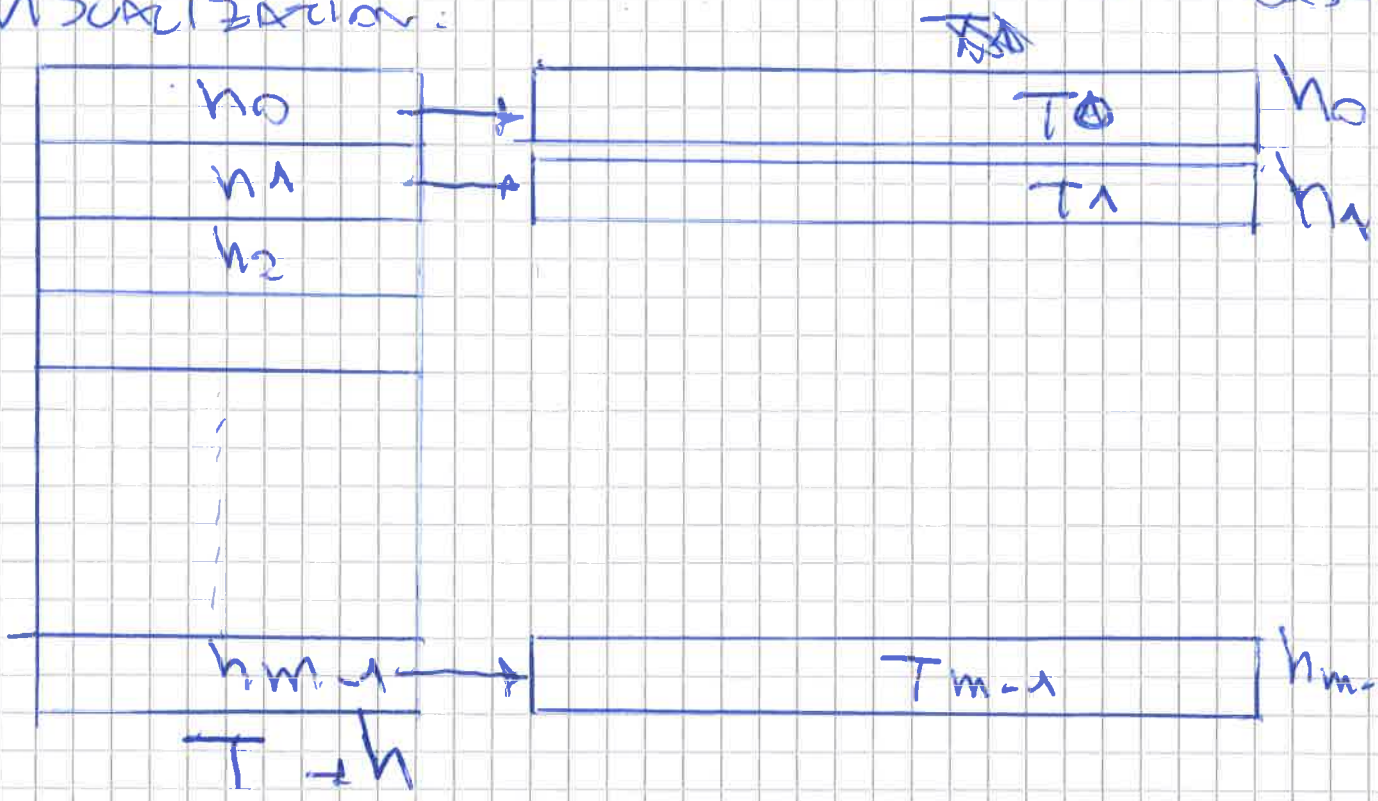
  ↓

- **SECOND-LEVEL HASHING**: Every entry of the first hash table points to a SECONDARY HASH TABLE T_j through an own UNIVERSAL HASH FUNCTION h_j.

⇒ Storing a key will require 2 TABLE ACCESSES & usage of 2 HASH FUNCTION.

→ h to T
→ h_i to T_j } O(1) worst case

VISUALIZATION:



We would like to ensure:

I) Space taken by T and all T_i's suitable in $O(n)$.

II) The hash functions used to map T to T_i's are PERFECT.

$q = n$

LEMMA:

If we store q keys in a HASH TABLE of size $m = q^2$, using a UNIVERSAL HASH FUNCTION, then we have that:

$$E\{\#\text{COLLISIONS}\} < \frac{1}{2} \quad \text{and}$$

$$E\{\#\text{COLLISIONS}\} \leq \frac{q^2}{2m}$$

PROOF:

In a set of q elements (keys), we have

$$E\{\#\text{COLLISIONS}\} = \binom{q}{2} < \frac{q^2}{2} \quad \text{because:}$$

$$\binom{q}{2} = \frac{q!}{(q-2)! \cdot 2!} = \frac{q \cdot (q-1)(q-2)!}{(q-2)! \cdot 2!} = \frac{q(q-1)}{2!}$$

$$\Rightarrow \frac{q \cdot (q-1)}{2} < \frac{q^2}{2} \quad \text{surely!}$$

If we pick h (a HASH FUNCTION) from a UNIVERSAL class of hash functions, then we have that each pair collides with $P_c = \frac{1}{m}$

$$E\{\#\text{COLLISIONS}\} = \binom{q}{2} \cdot \frac{1}{m} < \frac{q^2}{2m}$$

And since we have taken $m = q^2$

$$\Rightarrow E\{\#\text{COLLISIONS}\} = \binom{q}{2} \cdot \frac{1}{m} < \frac{q^2}{2m}$$

$$\Rightarrow E\{\#\text{COLLISIONS}\} < \frac{1}{2} \quad \text{Q.E.D.}$$

$\Rightarrow$ The PROBABILITY to have at least one COLLISION can be UPPER-BOUNDED via the MARKOV INEQUALITY:

$$P\{X \geq t \cdot E\{X\}\} \leq \frac{1}{t},$$

where in our case $t = 2$ and $X = \#\text{COLLISIONS}$.

And also, we have shown that

$$P\{\text{COLLISION}\} < \frac{1}{2} \Rightarrow P\{\text{NOT TO HAVE COLLISION}\} > \frac{1}{2}$$

$\Rightarrow$ We use this result to set

I) m_j of T_j as $m_j = (n_j)^2$ (SIZE of SECONDARY HASH TABLE (square of # keys in $T[j]$))

(SIZE of PRIMARY HASH TABLE)

II) $m = n$ for T to have that every hash function is perfect.

$\rightarrow O(n)$ SPACE. (WELL, PROOF)

THEOREM - BOUNDING THE SPACE OF PRIMARY & SECONDARY HASH TABLES. $\Theta(n)$

If we store n keys in a hash table of size $m = n$ using a UNIVERSAL HASH FUNCTION h, then the expected size of all sub-tables (SECONDARY HASH TABLES) T_j is $< 2n$.

PROOF.

$\Rightarrow$ We operate according to the following procedure when designing a PERFECT HASH TABLE.

First-level HASH TABLE

(1) Distribute keys of D into (1) by a UNIVERSAL HASH F.

(2) If $\sum_{i=0}^{n-1} (n_i)^2 \geq 2n$, then GOTO (1) and chose another hash function

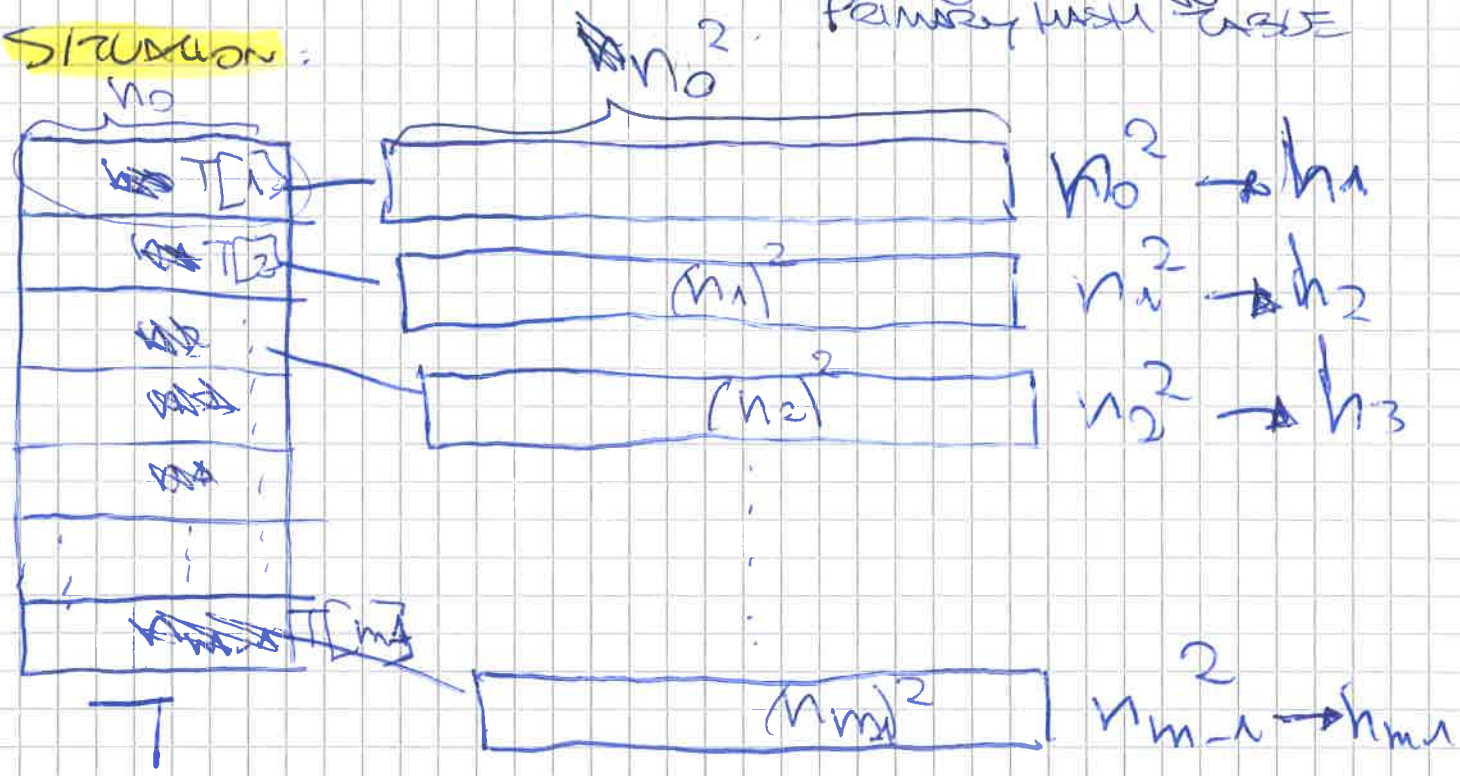
else (2) does not hold [If we have too many collisions]

$$\sum_{i=0}^{n-1} (n_i)^2 < 2n$$

[We picks a different hash function]

where n_i = #keys hashing to slot (1) of PRIMARY HASH TABLE

SITUATION:



The different slots pick their own hash function in an INDEPENDENT manner.

⇒ Let's hence estimate the #COLLISIONS in the different slots

$$\sum_{i=0}^{n-1} (n_i)^2 = \sum_{i=0}^{n-1} (n_i) + 2 \cdot \binom{n_i}{2}$$

$$E\left\{\sum_{i=0}^{m-1} (n_i)^2\right\} = E\left\{\sum_{i=0}^{m-1} (n_i) + 2\binom{n_i}{2}\right\} = n^2$$

Because: $V^2 = V + 2 \cdot \binom{V}{2} = V + 2 \cdot \frac{V!}{(V-2)!\,2!}$

$= V + V(V-1)$

$= V + V^2 - V$

$$E\left\{\sum_{i=0}^{m-1} (n_i)^2\right\} = E\left\{\sum_{i=0}^{m-1} n_i\right\} + 2E\left\{\sum_{i=0}^{m-1} \binom{n_i}{2}\right\}$$

total # items inserted (SECONDARY HASH TABLES)

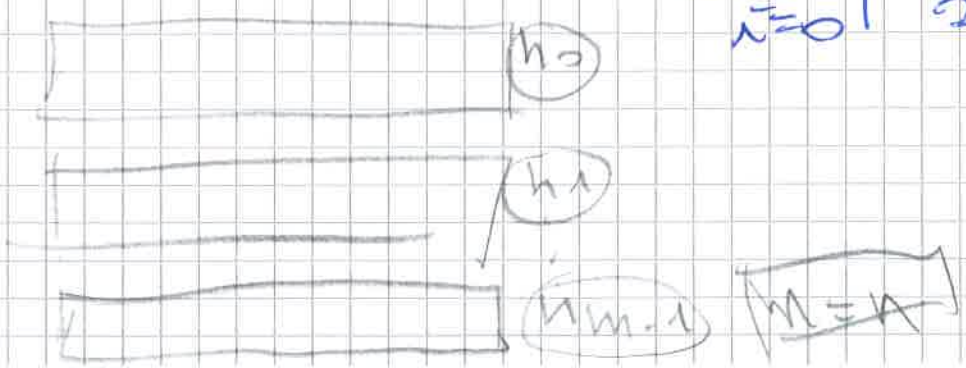
$$= n + 2 \cdot E\left\{\sum_{i=0}^{m-1} \binom{n_i}{2}\right\}$$

What is $\binom{n_i}{2}$?

⇒ It's the total # collisions caused by PRIMARY-LEVEL hash function (here consider every different pair)

$\sum_{i=0}^{m-1} \binom{n_i}{2}$ # collisions in:

$$\Rightarrow \sum_{i=0}^{m-1} \binom{n_i}{2} = \binom{n}{2} \frac{1}{m}$$



$$\Rightarrow \mathbb{E}\left\{\sum_{i=0}^{m-1} \binom{n}{i}\right\} = n + 2 \cdot \binom{n}{2} \cdot \frac{1}{n^m} \quad \Big| \; m = n$$

$$= n + 2 \cdot \left(\frac{n!}{(n-2)! \cdot 2!}\right) \cdot \frac{1}{n^n}$$

$$= n + 2 \cdot \left(\frac{n \cdot (n-1) \cdot (n-2)!}{(n-2)! \cdot 2!}\right) \cdot \frac{1}{n^n}$$

$$= n + 2 \cdot \frac{n \cdot (n-1)}{2} \cdot \frac{1}{n^n}$$

$$= 2n - 1$$

$$\Rightarrow \#\text{AVERAGE COLLISIONS} = 2n - 1 < 2n$$

$\Rightarrow$ We have hence shown that TWO-LEVEL HASHING is into bounded by $2n$.

$\Rightarrow$ On average, two iterations are SUFFICIENT to find an appropriate hash function. (PERFECT)

Also! We have shown that $\text{SPACE} < 2n$

$\Rightarrow$ SPACE $= O(n)$

CREATING A MINIMAL, ORDERED PERFECT HASH FUNCTION:

Recall:

MINIMAL HASH FUNCTION:

$\boxed{m = n}$ (i.e: no wasted space)

ORDERED HASH FUNCTION:

$\forall x < y$:

$$\boxed{h(x) < h(y)}$$

$\Rightarrow h = \text{Rank}_S(k)$

We want a function that, when given a key, it returns the Rank (i.e: the index) of it

	h_1	h_2	h (Rank)
t_1	1	6	0
t_2	7	2	1
t_3	5	7	2
t_4	4	6	3
t_5	1	10	4
t_6	0	1	5
t_7	8	11	6
t_8	11	9	7
t_9	5	3	8

h_1 and h_2 are two **UNIVERSAL HASH FUNCTIONS** (either given or can be arbitrarily chosen)

↳ Alternatively, one can just generate numbers for the table (randomly)

↳ Let's now consider also:

- φ: a function that maps integers in range $\{0, \ldots, m'-1\}$ to $\{0, \ldots, n-1\}$

We consider $m' = c \cdot n$, where $c \geq 1$ (two trivial cases: $c = 0$, $c = 1$)

$\Rightarrow$ φ must perform a PERFECT HASHING, or $m' \geq n$

We want to find t:

$$h(t) = \varphi(h_1(t)) + \varphi(h_2(t)) \bmod n$$

To do this, we INDIRECTLY CONSTRUCT an UNDIRECTED graph based on table T, where:

- Nodes are values of h_1 and h_2
- Edges connect the different nodes, and they are labeled by h the resulting hash value.

(2) Find values for the $\varphi(h_1(t))$ and $\varphi(h_2(t))$

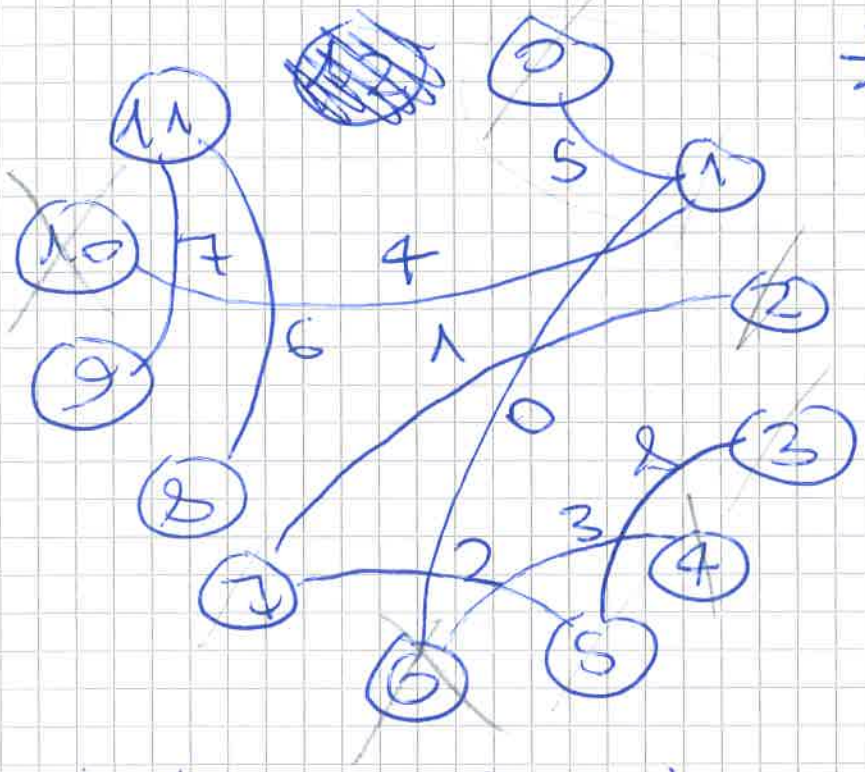
$\Rightarrow$ x: 0 1 2 3 4 5 6 7 8 9 10 11

φ: =

EXAMPLE. $C_0, \dots, m'-1$ $\quad m' = 12$,

(actually $m' = 13$)

$\#\text{NODES} = m' = 2 \cdot 9$



$h(t) = g(h_1(t)) + g(h_2(t)) \bmod n$

(RANK of t in S)

X:	0 1 2	3 4 5	6 7 8	9 10 11
g:	0 5 0	7 8 11	4 1 9	1 8 6

INIT: ~~We~~ We need to start somewhere!

① Start with $h_1(t) = 1$ and $h_2(t) = 6$

because $h_1(t) \to 1$ [RANK 1]

$h(t) = g(h_1(t)) + g(h_2(t)) \bmod n$

~~$2 = g(1) + g(6) \bmod 9$~~

$h[t] = g(h_1[t]) + g(h_2[t]) \bmod n$

Initially: select node 0, set $g(0)$ (need to store since.)

$g[1] = g(0) + g(1) \bmod 9$

0

$g[1] = 0 + g(1) \bmod 9$

$h[1] = g(h_1[1]) + g(h_2[1])$

We can set $g(1) = h(0,1) - g(0) \bmod n$

VALUE BEFORE

$h[t] = g(h_1[t]) + g(h_2[t]) \bmod n$

Initially: node 0, and set $\boxed{g(0) = 0}$

$h[0,1] = g(0) + g(1) \bmod 9$

$5 = 0 + g(1) \bmod 9$

$\Rightarrow g(1) = 5 \bmod 9$

$\Rightarrow g(1) = 5$

BUT $\Rightarrow$ Continue from node (1), as you have information about $g(1)$ there.

$h[1,0] = g(1) + g(0) \bmod 9$

$0 = 5 + g(0) \bmod 9$

$\Rightarrow g(0) = -5 \bmod 9$

$\Rightarrow \theta(6) = 4$

$h(6,4) = \theta(6) + \theta(4) \mod 9$

$3 = 4 + \theta(4) \mod 9$

$\theta(4) = -1 \mod 9$

$\Rightarrow \theta(4) = 8$

Let's continue with:

$h(10,1) = \theta(1) + \theta(10) \mod 9$

$4 = 5 + \theta(10) \mod 9$

$\theta(10) = -1 \mod 9$

$\theta(10) = 8$

(NO CONNECTED NODES left
→ Select a new node, like 2 and set.
$\theta(2) = 0$)

$h(2,7) = \theta(2) + \theta(7) \mod 9$

$1 = 0 + \theta(7) \mod 9$

$\Rightarrow \theta(7) = 1 \mod 9$

$\theta(7) = 1$

$h(7,5) = \theta(7) + \theta(5) \mod 9$

$2 = 1 + \theta(5) \mod 9$

$\theta(5) = 1$

$h(5,3) = \theta(5) + \theta(3) \mod 9$

$8 = 1 + \theta(3) \mod 9$

$g(3) = 5 \bmod 9$

$g(3) = 7 \bmod 9$

$\Rightarrow g(3) = 7$

<No modes left>

Set $g(0) = 0$

$h(0,11) = g(0) + g(11) \bmod 9$

$6 = 0 + g(11) \bmod 9$

$g(11) = 6 \bmod 9$

$g(11) = 6$

$h(11,9) = g(11) + g(9) \bmod 9$

$7 = 6 + g(9) \bmod 9$

$g(9) = 7 - 6 \bmod 9$

$g(9) = 1$

⇒ NB: After finding values for all $g(t)$, we can finally find $h(k)$ as:

$h(k) = g(h_1(k)) + g(h_2(k)) \bmod n$

$h(k_1, k_2)$

FINAL value

THEOREM — HASHING to SAME BUCKET

For any two distinct keys x and y, the P. Fenst x hashes to the same bucket as y is $O\left(\frac{1}{m}\right)$ (where m is the HASH TABLE's size.

CUCKOO HASHING. ⇒ DYNAMIC HASH TABLE!

Innovative idea for doing HASH TABLES' creation, maintenance (i.e.: update) and search.

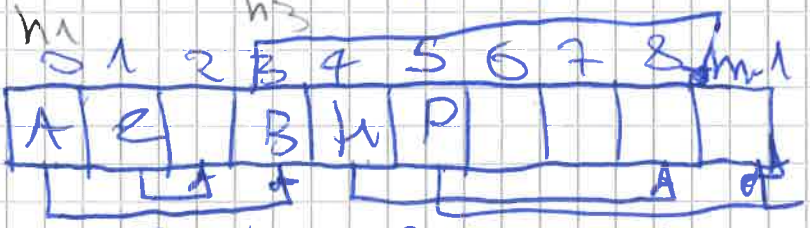
↳ WHY? It combines pre-existing ideas into literature to obtain:

- DELETION
  - $O(n)$ SEARCH TIME (WORST CASE)
  - $O(n)$ UPDATE TIME (AMORTIZED COST)
  - $O(n)$ SPACE

↳ CUCKOO HASHING ≡ $O(n)$

It is based on 2 HASH FUNCTIONS: h_1 and h_2 that operate by the "POWER OF 2" approach when inserting a key, yet with a "TWIST".

↳ Consider a HASH TABLE:



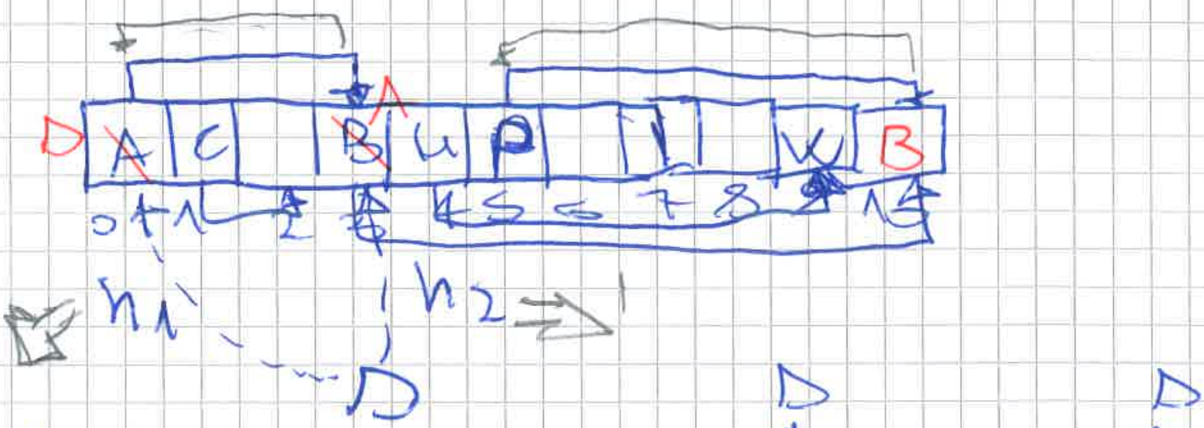
There are EDGES between keys in a random manner based on the HASHED VALUE of the two hash functions

$\forall$ key k:

$\text{edge}(h_1(k), h_2(k)) \Rightarrow$ MEANING: k could either be in $h_1(k)$ or $h_2(k)$

Two random cells are computed by a hash function!

INSERTION: Ex: We want to insert (D) and ~~[illegible]~~ ⇒ Operate by the "POWER OF 2" ARGUMENT



~~If A is empty~~ Check $h_1(k)$ and $h_2(k)$.
If either slot is empty insert D there.
If both slots are filled with another value, then ⇒ EVICT element in $h_1(k)$

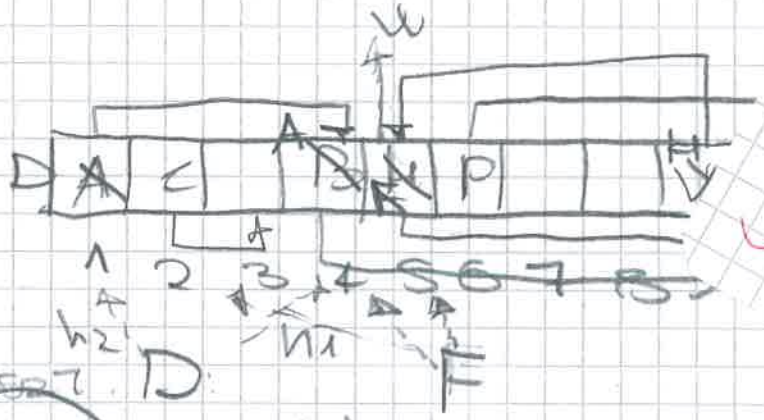
~~[illegible] Kick out A from (0)~~

⇒ Kick (A) out of slot [0] ⇒ Insert (D) into slot (0) ⇒ ~~[illegible]~~ Percolate edge from (0) to (3) ⇒ Kick (B) from slot (3) ⇒ Insert A into slot (3) ⇒ Percolate edge from (3) to (9) ⇒ Insert (B) into (9), so it's empty.

CASCADE of KEY MOVEMENTS

[We keep following edges till we find an empty slot] ⇒ If all cells are full, then we cannot ~~[illegible]~~ insert anything ↓

BOOK
EXAMPLE - CUCKOO HASHING "5"



Cuckoo Brutal Force

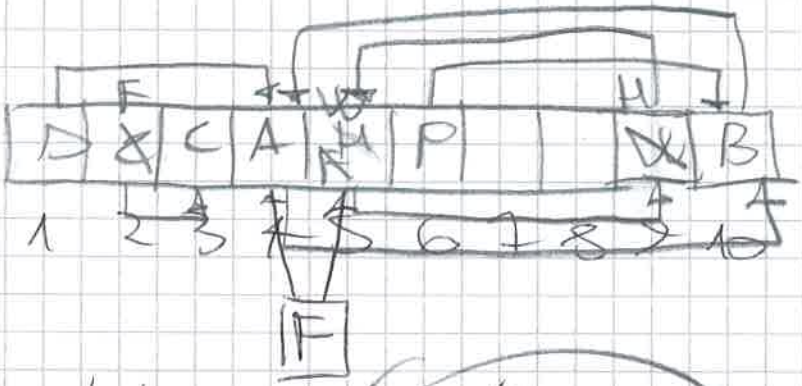
=> INSERT D:

$h_1(D) = 4$, $h_2(D) = 1$ => Both are occ.

=> Use pick (A) for eviction

=> INSERT F:

$h_1(F) = 4$; $h_2(F) = 5$



$h_1(F) = 4$, $h_2(A) = 5$

Causes "LOOPED" EVICTIONS
Must stop upon reaching (5) again. At that point we also check $h_1(K)$ for insertion

=> After evicting element, we need to INSERT the dropped

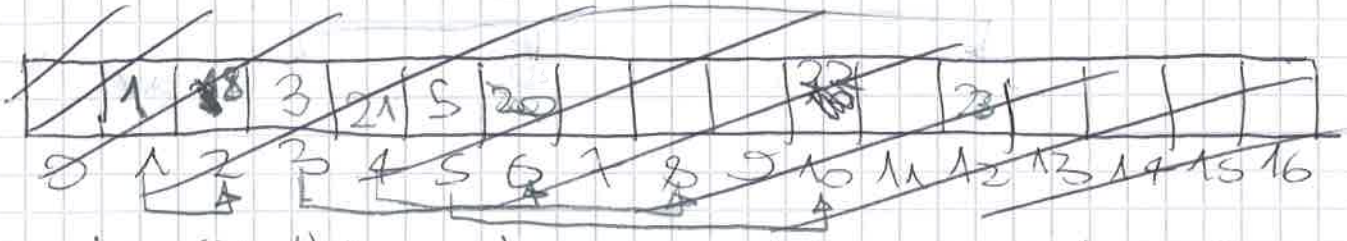
EVICTION: Firstly call h_1; After fully performing it, we will have an intermediate element, also check h_2.

CUCKOO HASHING – CLASS EXERCISE

Build a CUCKOO HASH TABLE based on the following two HASH FUNCTIONS.

$h_1(k) = k \mod 17$ ⇒ $m = 17$

$h_2(k) = 2k \mod 17$



- $h_1(k)$ taken, h_2 not taken ⇒ Insert on $h_2(k)$.
- $h_1(k)$, $h_2(k)$ ⇒ Need to check at $h_1(k)$.

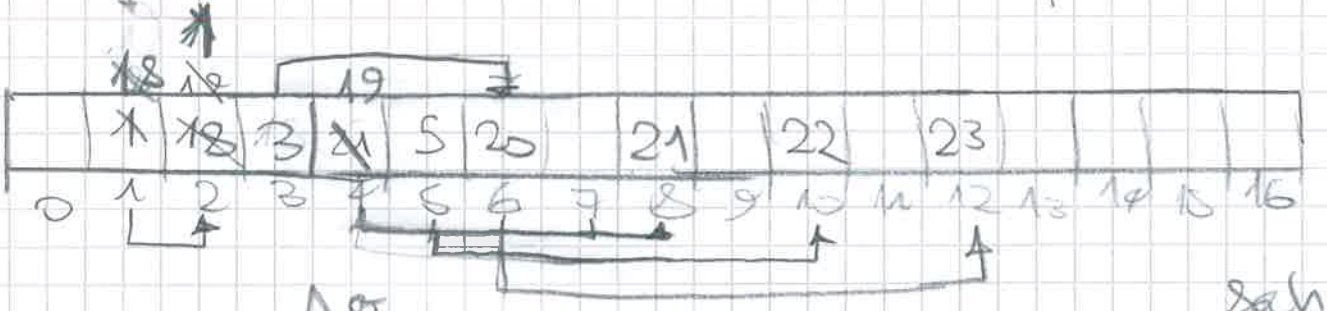
$S = \{1, 5, 18, 3, 22, 20, 21, 23, 19\}$

$h_1(19) = 2$

$h_2(19) = 4$

$h_1(k)$ $h_2(k)$

$\Theta(h_1(k), h_2(k))$



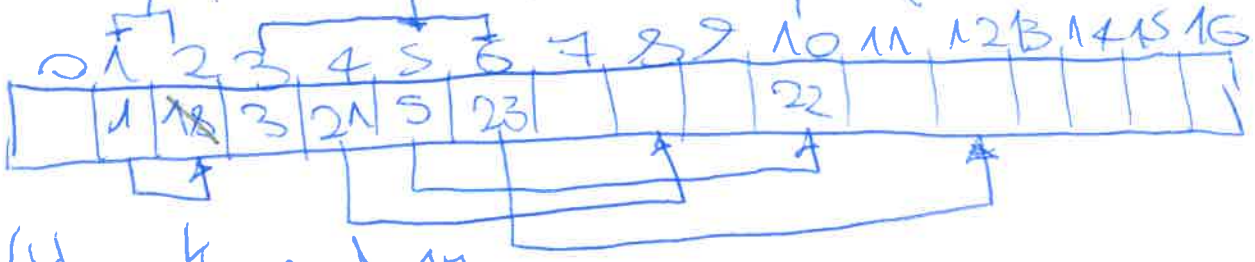
After

NB: performing "CYCLED" EVICTIONS, we need to check h_2

⇒ Also, we need to update edges' direction!

CUCKOO HASHING - EXERCISE WITH FRANCESCO:

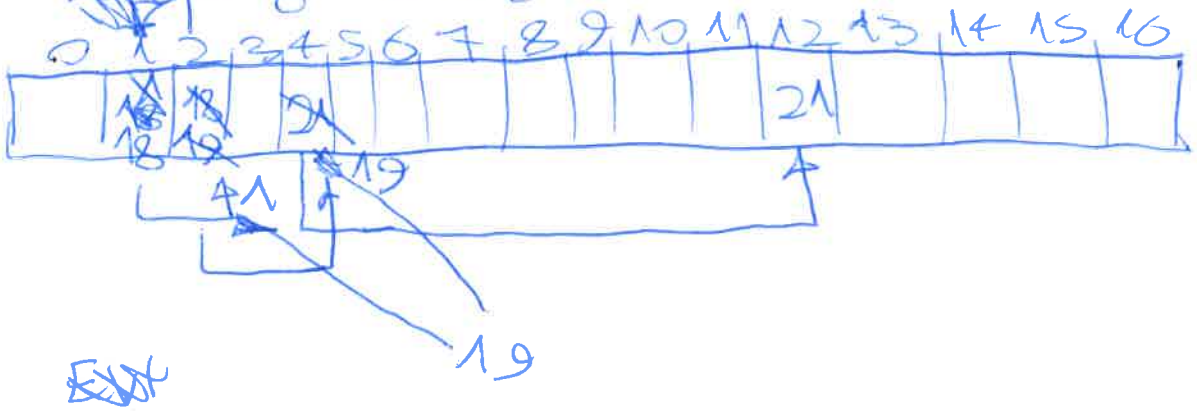
$S = \{1, 18, 3, 21, 5, 23, 22, 19\}$



$h_1(k) = k \bmod 17$

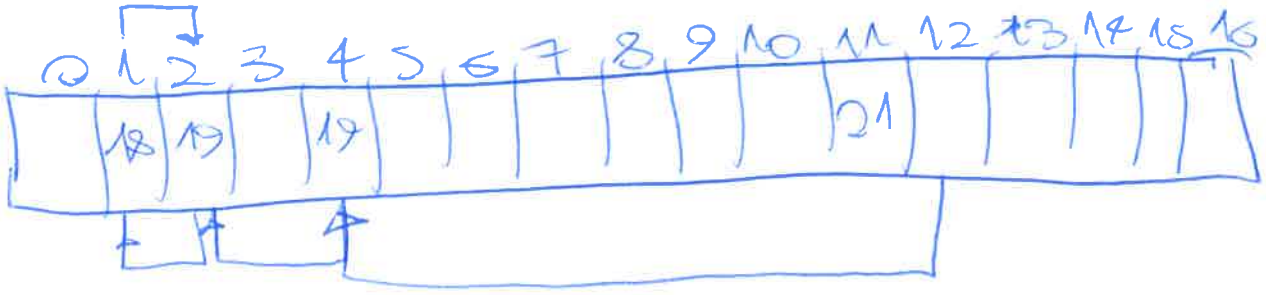
$h_2(k) = 2k \bmod 17$

Everything alright till 19 ⇒ NOW EVICT:

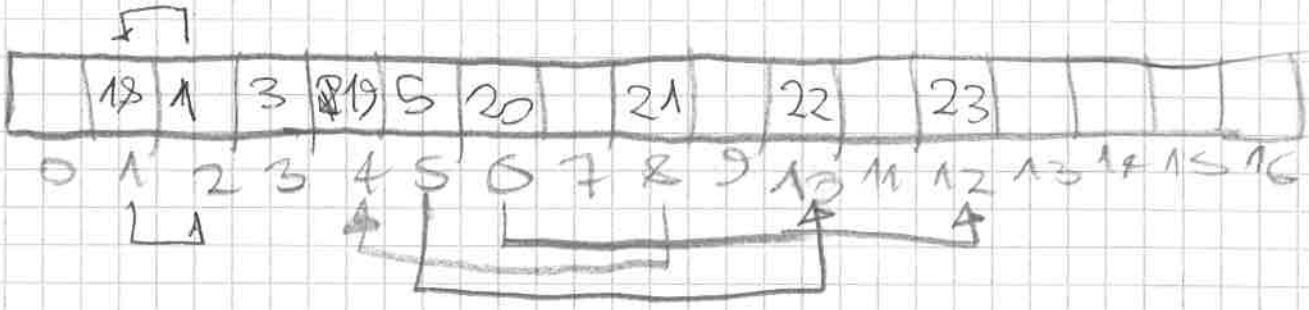


EXIT

And now INVERT the EDGES:



RESOLVING HASH TABLE.



THEOREM - HASHING TO THE SAME BUCKET IN CUCKOO HASHING:

For any two DISTINCT keys x and y, the $P\{x$ hashes to the same bucket as $y\} = O\left(\frac{1}{m}\right)$

Proof:

If x and y are in the SAME bucket, then there is a PATH of some length L between one node in $\{h_1(x), h_2(x)\}$ and one node in $\{h_1(y), h_2(y)\}$

$\Rightarrow$ The same claim by PREV. THEOREM that $\forall$ entries in y and any constant $c > 1$, if $m \geq 2 \cdot c \cdot n$, then:

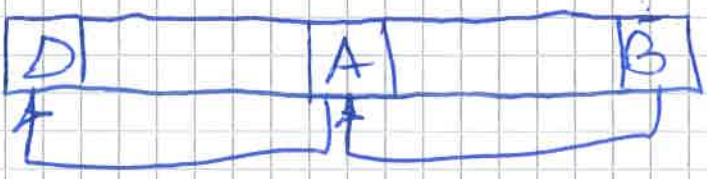
$$P\{i \rightsquigarrow j\} = \frac{c^{-L}}{m}$$

$$\Rightarrow 4 \cdot \sum_{L=1}^{\infty} \frac{c^{-L}}{m} = \frac{4}{m} \sum_{L=1}^{\infty} \left(\frac{1}{c}\right)^L = \frac{4}{m} \cdot \frac{\left(\frac{1}{c}\right)}{1 - \frac{1}{c}}$$

$$= \frac{4}{m} \cdot \frac{\left(\frac{1}{c}\right)}{c-1} = \frac{4}{(c-1) \cdot m} = O\left(\frac{1}{m}\right) \quad \text{Q.E.D.}$$

NB: After inserting new values, we need to INVERT the EDGES' DIRECTION of the ones previously following an EVICTION

Ex:



CYCLES? HOW TO DEAL WITH THEM:

↳ It can happen that hash functions lead to the creation of CYCLES among keys!

↳ • 1 CYCLE: N.P.! → Perfect Pairings!

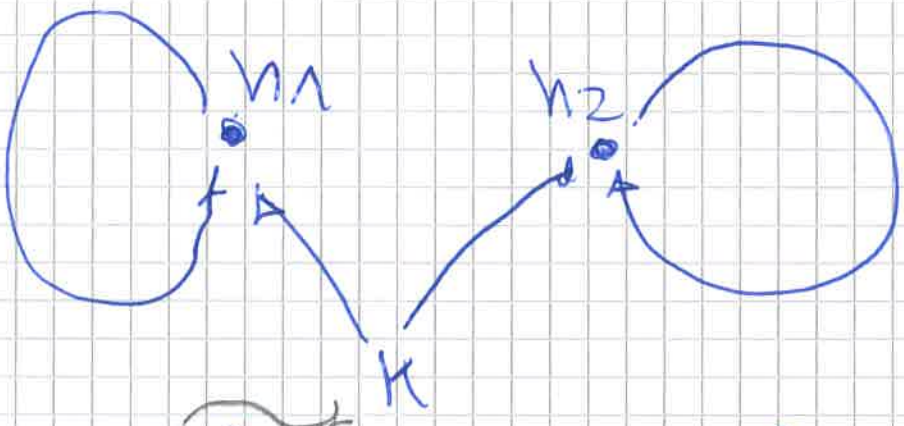
• 2+ CYCLES: Issues!

↳ REHASH all elements

Need to pick new hash functions and re-hash all the DICTIONARY KEYS

(However, this occurs with bounded P.)

→ Cost of re-hashing can be amortized by cost of O(1) for inserting.



PROOF. Having one cycle is very improbable (i.e.: The O(n) cost of re-hashing is AMORTIZED by insertion taking O(1) time.

Ex: The 22.000 € of buying a new VW Polo is amortized over 10 years.

THEOREM: L = Maximum length between any i and j (nodes)

If m = array size = #NODES

n = #KEYS = # EDGES

LENGTH of a PATH:

$\forall i,j$ entries in T (hash table) and any constant $c > 1$.

If $m \geq 2 \cdot c \cdot n$ (at most ...)

$$P\{i \xrightarrow{L} j\} \leq \frac{c^{-L}}{m} = \frac{1}{m \cdot c^{L}}$$

Pattern i to j

⇒ two nodes are connected by a path of length L or all hashing

i.e.,

the Probability to generate a path gets always smaller and smaller (exponentially).

Proof – By induction over L and L:

- **BASE CASE** $L = 1$ (i.e. one edge between i and j)

$P\{i \xrightarrow{1} j\}$ = Path long 1 (1 edge)

$= \frac{1}{m}$ (h_1 random)

$\leq \sum_{k=1}^{m} \frac{2}{m^2} = \frac{2m}{m^2} = \frac{2}{m} \leq \frac{1}{2 \cdot c \cdot n}$ (h_2 random among m values)

edge between i and j — all keys

UNION BOUND

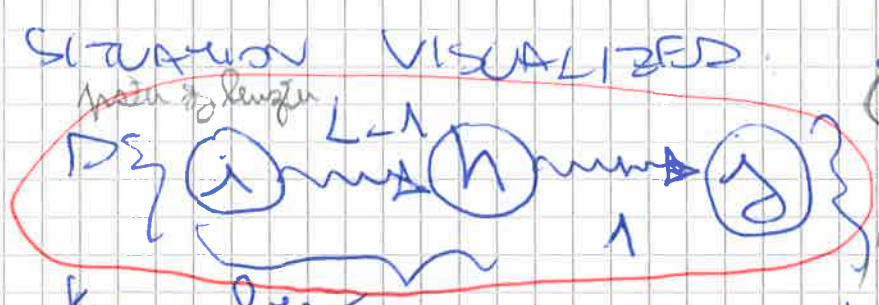
Hyp: $m \geq 2 \cdot c \cdot n \Rightarrow n \leq \frac{m}{2c}$

$$\Rightarrow P\{i \xrightarrow{1} j\} \leq \frac{1}{cm}$$

- **INDUCTIVE CASE** $L > 1$

Assume the bound holds for $(L-1)$. We want to show that it holds for L as well.

SITUATION VISUALIZED



Known by INDUCTIVE HYPOTHESIS

$\leq \frac{1}{m} \cdot \frac{1}{C^{L-1}}$ — remaining EDGE $(h \to j)$

$\leq \frac{1}{m} \cdot \frac{1}{C^{L-1}}$ — INDUCTIVE HYP $(i \to h)$

$\leq \frac{1}{m} \cdot \frac{1}{C^{L-1}} \cdot \frac{1}{C} = \frac{1}{m \cdot C^{L}}$ Q.E.D.

(NB: By INDUCTIVE HYPOTHESIS, we had that

$P\{i \xrightarrow{L-1} j\} \leq \frac{1}{C^{L-1} m}$

PROBABILITY of CYCLES EXISTENCE:

We now want to prove that:

$P\{\exists\ i \to \dots \to i\} \leq \sum_{L=1}^{+\infty} P\{i \xrightarrow{L} i\} \leq \sum_{L=1}^{+\infty} \frac{1}{m \cdot C^{L}}$

(There exists a cycle on i)

Consider all the possible paths' lengths leading to a cycle

$= \frac{1}{m} \sum_{L=1}^{+\infty} \left(\frac{1}{C}\right)^{L} = \frac{1}{m} \cdot \frac{\frac{1}{C}}{1-\frac{1}{C}} = \frac{1}{m} \cdot \frac{\frac{1}{C}}{\frac{C-1}{C}} = \frac{1}{m(C-1)}$

$\Rightarrow P\{\text{cycle on } i\} \leq \frac{1}{m} \cdot \frac{1}{C-1}$

All possible i

$P\{\text{CYCLE EXISTENCE}\} \leq m \cdot \frac{1}{m} \cdot \frac{1}{C-1}$

$P\{\text{CYCLE}\} \leq \frac{1}{C-1}$

Upper bound to the P. of unsuccessful insertion

Corollary 1:

We have found that:

$$P\{\text{cycle}\} \leq \frac{1}{c-1}$$

$\Rightarrow$ We would like $\frac{1}{c-1} = \frac{1}{2} \Rightarrow 2 = c-1$

$\Rightarrow c = 3$

Set $c \geq 3$, and we have that:

$$m \geq 2c \cdot n$$

$$\Rightarrow m \geq 6 \cdot (1+\varepsilon)\, n$$

- Taking a CUCKOO TABLE of this size, the P. to have a cycle in the cuckoo graph of the original dictionary is $\leq \frac{1}{2}$

$$\Rightarrow P\{\text{CYCLE}\} \leq \frac{1}{2}$$

$\Rightarrow$ A constant # REHASHES is enough to ensure that the insertion of Σn keys in a dictionary of size n. $\Rightarrow$ One rehashing takes $O(n) \Rightarrow O(\frac{1}{\varepsilon})$ per INSERTION

Corollary 2:

By setting $c \geq 2$ and taking a cuckoo table of size $m \geq 2\varepsilon n\,(1+\varepsilon)$, the cost of inserting Σn keys in a DICTIONARY of size n is constant AMORTIZED (AMORTIZED as the cost of $\Theta(n)$ INSERTION & RE-HASH).

~~Let's Proof~~:

We want to show that, if the #

$\#keys \leq (1+\varepsilon)n \Rightarrow$ Theorem holds!

Let's build a HASH TABLE of ~~size~~ $m = 6 \cdot \varepsilon k n_0$ (i.e.: larger than necessary!)

Let's consider:

$P\{\text{fail at } T_0 \text{ insertion}\} \leq P\{\text{fail at } T_0+1\}$

$\Rightarrow P\{\text{table reconstruction at } T_0\} \leq P\{\text{table reconstruction at } T_0+1\}$

$\rightarrow$ Always increasing P. to fail

$\Rightarrow$ Let's now consider the time instant T_1.

LAST INSERTION

$\Rightarrow P\{\text{fail at } T_1\} \leq \frac{1}{2}$

[On average two reconstructions are enough at T_1]

$\Rightarrow$ The AMORTIZED COST no change!

~~[i.e. two reconstructions]~~

(Each INSERTION would take $\frac{1}{\varepsilon}$)

GLOBAL REBUILDING TECHNIQUE

Analogously to what we did with chaining, if the size of the dictionary becomes too small compared to the size of the hash table, a new smaller hash table is created;
(save space)

If the size of the dictionary becomes too large with respect to the hash table it is filled up with all nodes of the dictionary, a new larger hash table is re-created (double, or by a constant factor).

K HASH FUNCTIONS. [MULTI-CASE]

Consider K hash functions instead of 2.

$h_1, h_2, \ldots, h_K$

⇒ We have a tree like approach for every cell (bag) ⇒ COMPLEX VISUALIZATION

BUCKET-LIKE APPROACH.

Every cell is a BUCKET (we no longer have cells!)

We can define a LOAD FILLING FACTOR

BLOOM FILTERS

Data structure used to store a very LARGE universe of keys, hence occupying a large space [Ex: URLs managed by crawlers in search engines]. Very long!

⇒ Need to keep them in MEMORY!

IDEA: Do not store strings themselves, but a simple and randomized representation of them (FINGERPRINT) in a BINARY VECTOR

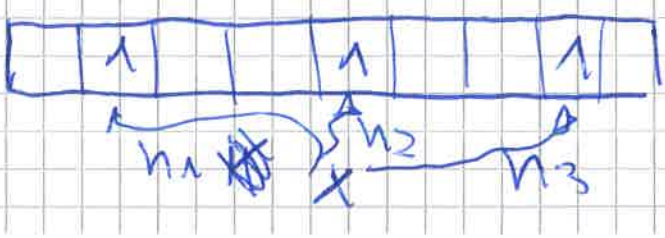
⇒ DRAWBACK: One-sided-error, yet error rate decreases exponentially with the size of the FINGERPRINT of keys.

$\forall K \in D \Rightarrow$ Answer is yes [NO ERROR]

$K \notin D \Rightarrow$ Answer is ~~NO~~ (MAYBE) YES / NO

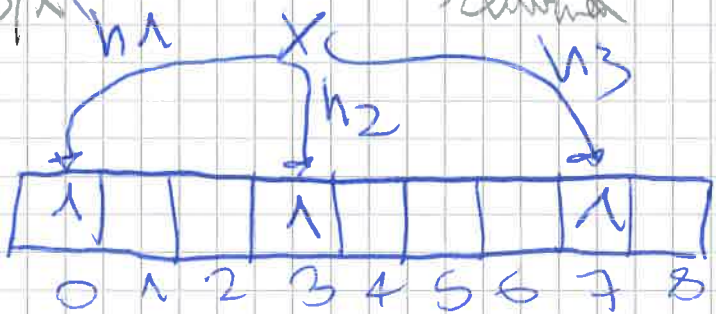
⇒ There may occur FALSE POSITIVES in the search, in case elements are NOT in the dictionary D.

⇒ SIMPLE, RANDOMIZED ACCESS via hash onto HASH FUNCTIONS:



FUNCTIONS:

- INSERT(x) $\Rightarrow$ Set all bits produced by hash functions $h_1, \dots, h_n$ to 1. — $O(1)$
- SEARCH(x) $\Rightarrow$ YES $\longleftrightarrow$ All bits of $B[h_i(x)]$ are 1. — $O(1)$ — returned



My FALSE POSITIVES are POSSIBLE!

SEARCH: Re-compute hash functions of x and see if ALL those bits are 1. $B[h_i(x)]$

FALSE POSITIVE: However, those bits may have been set to 1 by another hash function!

$\Rightarrow$ The P. of error depends on k (i.e. the # HASH FUNCTIONS used) (the more the merrier)

ESTIMATION of the P. of FALSE POSITIVE

(i.e.: P[a query yields a FALSE POSITIVE])

COUNTING ARGUMENT: (count # cases in which we have a FALSE POSITIVE)

$P[$ bit is still 0 after inserting n keys $]$

$= P[k$ independent hash functions $H_i(k)$ returned any entry different from $i]$

$= \left(\frac{m-1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$

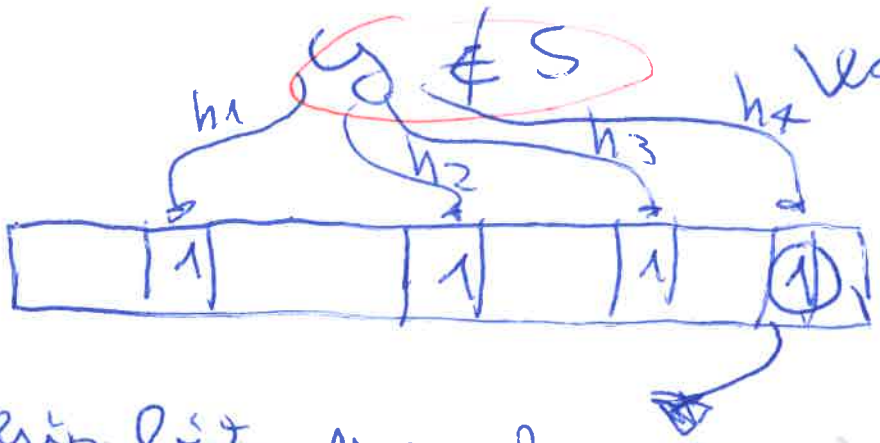
ESTIMATION of the P. of FREQ in a BLOOM FILTER

- TRUE NEGATIVE:

CORRECT! ⇒ y is not detected to be in S

$y \notin S$

- FALSE POSITIVE:



WRONG ⇒ y is detected to be in S, though it is actually not in S.

This bit may have been set to 1 because of other hashes of other keys.

$P\{$ FALSE-POSITIVE ERROR in a BLOOM FILTER $\} = ?$

PROOF: (of the P. of FALSE POSITIVE in a BLOOM FILTER)

$P\{$ The insertion of a key left $B[i] \Rightarrow$  $\} = P\{$ k independent hash functions returned an entry $\neq i\}$

because of i

$= \left(\frac{m-1}{m}\right)^k = \left(1-\frac{1}{m}\right)^k$

After inserting n many keys, the P. test $B[i]$ to still 0 is:

$= \left(1 - \frac{1}{m}\right)^{n \cdot K}$

$P\{$Bit $B[i]$ no 1 after inserting n items$\}$

$= \left(1 - e^{-\frac{n \cdot k}{m}}\right)$

$= \left(1 - \frac{1}{m}\right)^{\frac{n \cdot k \cdot m}{m}} \simeq e^{-\frac{nk}{m}}$

$x \notin D$, answering "yes" for $m \gg 1$

$P\{$FALSE-POSITIVE ERROR$\} =$ ~~all k bits checked~~

$= P\{$all K bits checked for a key not in the current dictionary are set to $1\}$

$= P\{\forall x : B[h_i(x)] = 1\} = \left(1 - e^{-\frac{nk}{m}}\right)^K$

$\Rightarrow$ ~~Perr~~ $P_{err} = \left(1 - e^{\frac{nk}{m}}\right)^K$

Where the error probability depends on:

- k = # HASH FUNCTIONS (the higher, the lower the Perr)
- m = # BITS in the BINARY ARRAY (the more, the better)
- n = # DICTIONARY KEYS (the fewer, the better)

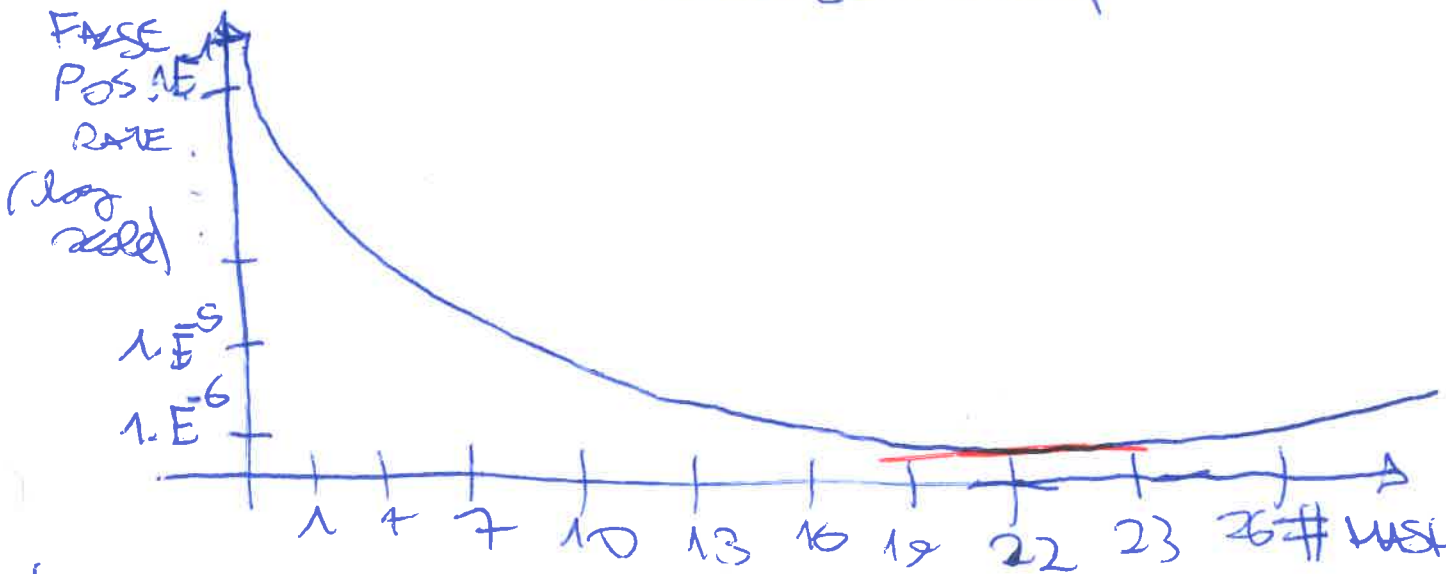
NB: $g = \frac{m}{n}$ = AVG. # BITS per dictionary key allocated in B (fingerprint size)

↳ The larger g, the smaller the ERROR PROBABILITY, but the larger the SPACE allocated

OPTIMAL

OPTIMAL VALUE FOR k

k = # HASH FUNCTIONS



If we fix m and n & #keys.

↳ BLOOM FILTER'S SIZE.

what is the OPTIMAL value for k.

(i.e: the one value minimizing perr)

We know that:

$$P\{x \notin D, \text{answering "YES"}\} = \left(1 - e^{\frac{nk}{m}}\right)^k$$

OPTIMAL k: (# HASH FUNCTIONS)

We compute $\frac{\partial}{\partial k}(\cdot) = 0$ to find the MINIMUM.

$$k = \frac{m}{n} \cdot \ln 2 = \left(\frac{1}{2}\right)^{\frac{m}{n}} \cdot \ln 2$$

$$\Rightarrow (0.6185)^{\frac{m}{n}}$$

Longer k ⇒ more 1s in B

Smaller k ⇒ more 0s in B.

⇒ For OPTIMAL k ⇒ Same # 0s & 1s in B!

$$P_{ERR} = (0.6185)^{\frac{m}{n}}$$

FOR k OPTIMAL

SPACE & BLOOM FILTER:

↳ LOWER-BOUNDED by:

$$m \geq \frac{n \cdot \log_2\left(\frac{1}{\varepsilon}\right)}{\ln 2} \approx 1.44 \cdot n \cdot \log_2\left(\frac{1}{\varepsilon}\right)$$

$\Rightarrow$ SPACE = $O(n)$ = OPTIMAL!

(like ~~HASH~~)

↳ To store (n) ~~KEYS~~ keys

UNIVERSAL HASH FUNCTIONS - CLASS EXERCISE.

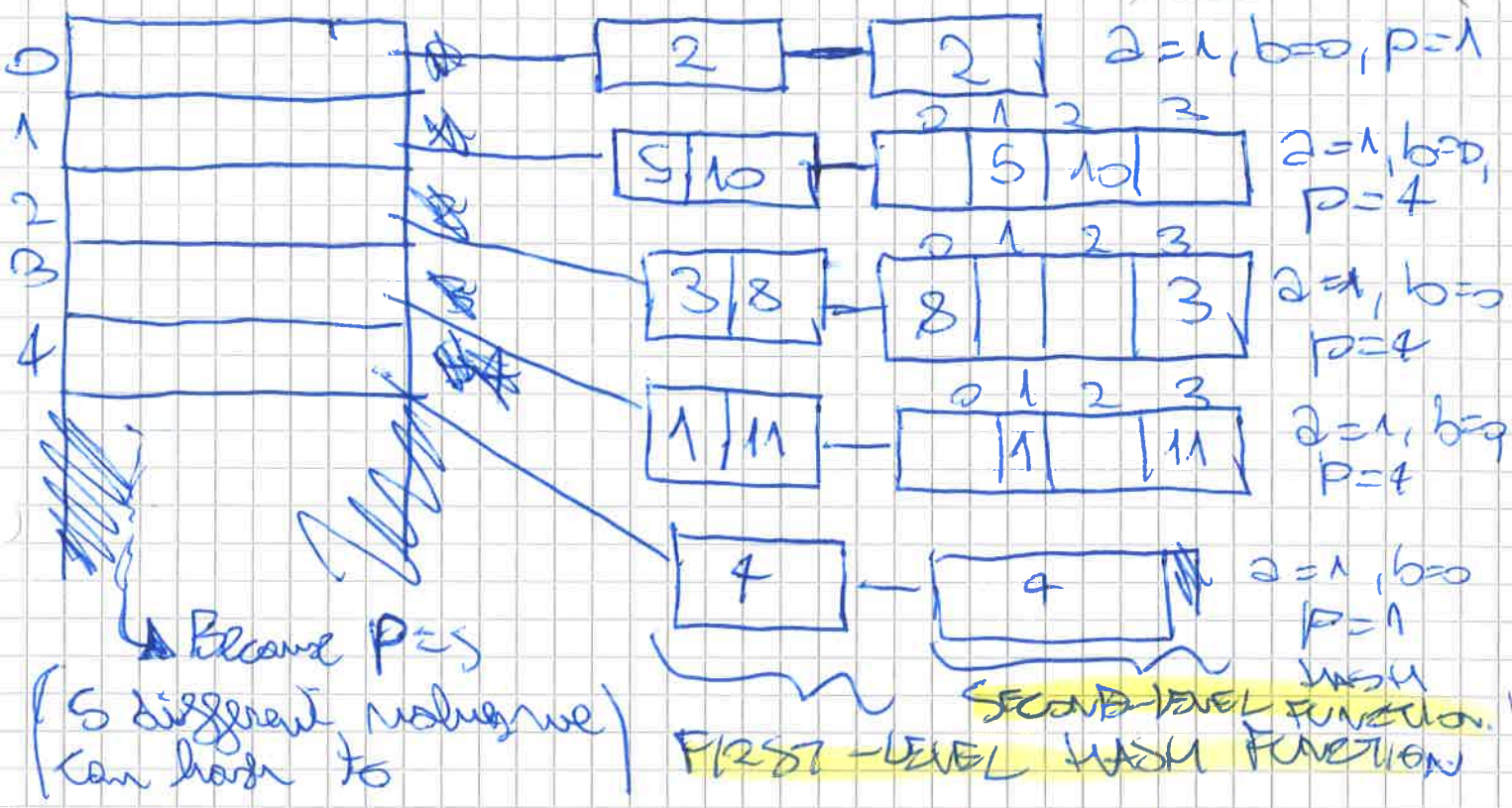
$S = \{1, 2, 11, 13, 15, 18, 21, 23\}$

$$h_{a,b}(x) = ax + b \bmod p$$

First-level hash function:

$h_{1,0} = x \bmod 11$ (p)

$h_{2,1} = 2x + 1 \bmod 5$



Is it perfect? (i.e. no collisions).

$$\sum_i (n_i)^2 = 14 < 2n \quad (n = 7)$$

(n_i = # elements in bucket i)

[Otherwise, would need a different hash function!]

EX: Want to check if a number is contained in the hash table.

$\Rightarrow x = 7$

1) $h_1 = 2x + 1 \bmod 5$

$h_1 = 0 \Rightarrow$ Go to BUCKET ⓪.

2) $h_2 = x \bmod 1$

$\Rightarrow$ NOT PRESENT!

In order to pick a UNIVERSAL HASH FUNCTION generally 2 trials are sufficient.

$h_{a,b}(x) = ax + b \bmod p$,

where $p \rightarrow m$

↳ PRIME NUMBER.

MINIMAL ORDERED PERFECT HASH FUNCTION.

$S = \{aa, ab, bb, bc, ca, db\}$

rank
a = 1
b = 2
c = 3
d = 4

And we are given:

$h_1(x,y) = rank(x) \cdot rank(y) \bmod m$

$h_2(x,y) = rank(x) + rank(y) \bmod m$

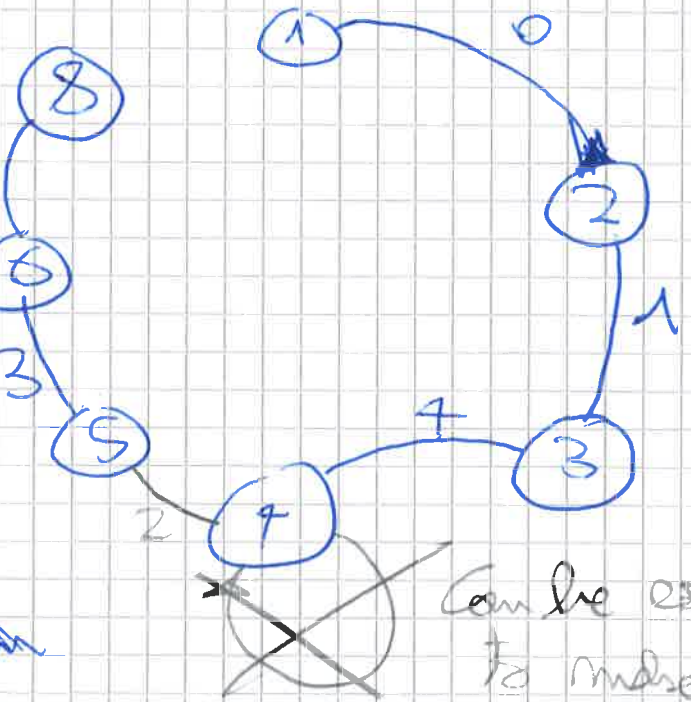
~~$m' = 2m$~~

And also: $m' = 11$;

$h = 6$

We compute the respective HASH FUNCTIONS

	h_1	h_2	rank
aa	2	1	0
ab	2	3	1
bb	4	4	2
bc	6	5	3
ca	3	4	4
db	8	6	5



Can be REMOVED to maintain graph CYCLE FREE

We notice that ~~we~~ have as many nodes as the # different hash values.

⇒ We now need to find values for g_j:

g_j	X
0	1
0	2
1	3
3	4
5	5
4	6
1	8

NB: This works only for strings actually inserted in the hash table!
Not for other strings.

- $h(1,2) = g(1) + g(2) \bmod 6$
  $\Rightarrow 0 = 0 + g(2) \bmod 6$
  $g(2) = 0 \bmod 6$
- $h(2,3) = g(2) + g(3) \bmod 6$
  $1 = 0 + g(3) \bmod 6$
  $g(3) = 1 \bmod 6$
- $h(3,4) = g(3) + g(4) \bmod 6$
  $4 = 1 + g(4) \bmod 6$
  $g(4) = 3 \bmod 6$
- $h(4,5) = g(4) + g(5) \bmod 6$
  $2 = 3 + g(5) \bmod 6$
  $g(5) = -1 \bmod 6$
  $g(5) = 5$
- $h(5,6) = g(5) + g(6) \bmod 6$
  $3 = 5 + g(6) \bmod 6$
  $g(6) = -2 \bmod 6$
  $g(6) = 4$
- $h(6,8) = g(6) + g(8) \bmod 6$
  $5 = 4 + g(8) \bmod 6$
  $g(8) = 1$

BLOOM FILTER - CASS EXERCISE:

1/ Given m and $n = 2^{16}$ # keys to insert

m = Bloom filter's size

$\Rightarrow$ Compute the OPTIMAL # HASH FUNCTIONS to use & the ERROR.

k_{opt} = OPTIMAL # HASH FUNCTIONS $= \frac{m}{n} \ln 2$

$= \frac{2^{20}}{2^{16}} \ln 2 = 2^4 \ln 2$

$\varepsilon_{OPT} = (0.6185)^{\frac{m}{n}} = (0.6185)^{2^4} = (0.6185)^{16}$

ERROR

2/ Compute the size of BLOOM FILTER (m) to guarantee an error of $\varepsilon_m = 2^{-20}$ by using an OPTIMAL # HASH FUNCTIONS & knowing we have to insert $n = 2^{16}$ keys.

$k_{opt} = \frac{m}{n} \ln 2$

$\varepsilon = (0.6185)^{\frac{m}{n}} \Rightarrow 2^{-20} = (0.6185)^{\frac{m}{2^{16}}}$

$\Rightarrow$ After finding m, we can then find k_{opt}.

$2^{-4} = (0.6185)^{\frac{m}{2^{16}}}$

$m = 5,78 \quad k_{opt} = \frac{5,78}{2^{16}} \ln 2 = ?$

$$error = \left[1 - e^{-\frac{kn}{m}} \right]^k$$

SEARCHING STRINGS BY PREFIX

CHAPTER 9

PREFIX SEARCH

PROBLEM STATEMENT:

INPUT: • Dictionary D consisting of N strings.

• Query P = A string containing a set of characters.

OUTPUT: Strings of D that have P as a PREFIX.

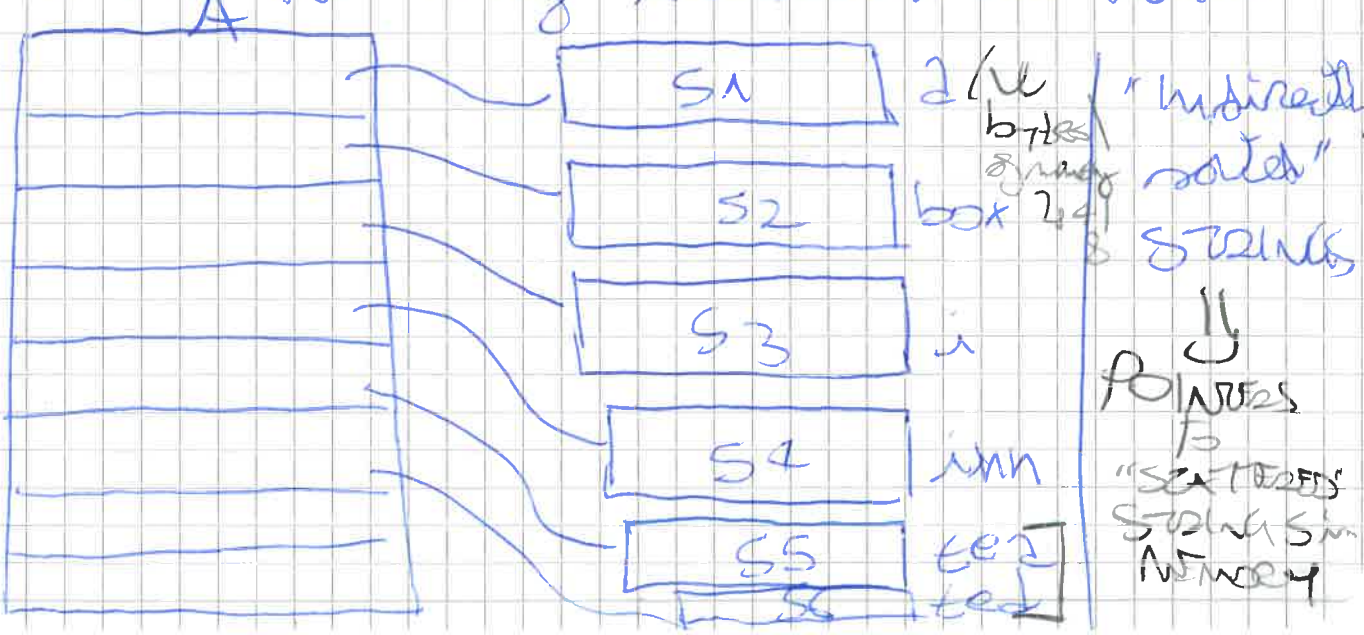
n = # STRINGS in D.

N = Total size of D.

P = [1, P] = Array of (P) characters.

(1) SOLUTION – ARRAY of STRING POINTERS

We take an INPUT: Array A of pointers storing arbitrary locations on disk.



A[1..N] is the result of calling malloc for (N) strings.

↳ Arbitrary strings' allocation, in fact memory management is delegated to the SYSTEM.

BASIC APPROACH'S IDEA:

Perform a simple [binary search] of (P) over the STRINGS' PREFIXES in the dictionary.

COMPARISON STEPS $= O(\log_2 N)$

COST of 1 STRING COMPARISON $= O(P)$ → in worst case, scan all the P characters

⇒ [PREFIX BINARY SEARCH COST $= O(P \cdot \log_2 N)$]

SPACE $= O(N + n)$

Ex: P = te

⇒ We are interested in finding tea, ted!

↓

Goal: Identify BLOCK's starting and ENDING POSITION of the leds of strings where tea and ted are located. [i.e. of all strings having that prefix]

WE EXPLOIT 2 MAIN PROPERTIES for this BASIC APPROACH:

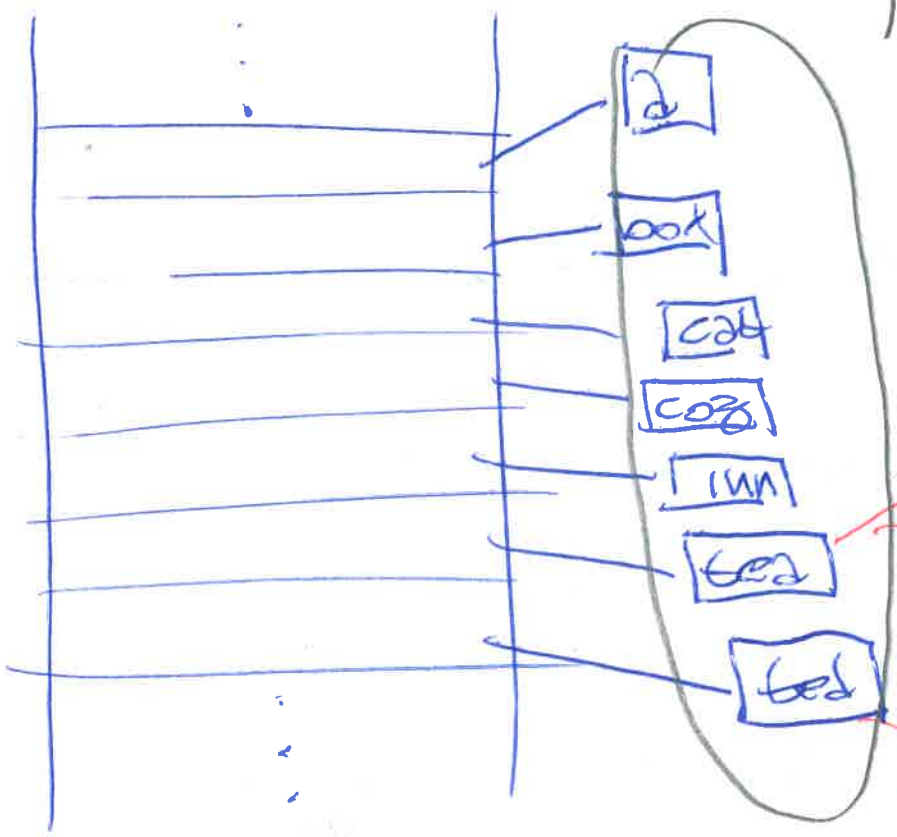
1) All DICTIONARY strings prefixed by pattern P occur contiguously if they are LEXICOGRAPHICALLY SORTED, for their pointers occupy a CONTIGUOUS sub-array, say A[l, r]

starting position, ending position.

NB: A[l, r] may be empty if no strings are prefixed by P.

2) String P is lexicographically located between A[l-1] and A[r].

EXAMPLE: P = "te"



NB: These STRINGS could be ANYWHERE in memory. POINTERS (malloc)

1-BASIC APPROACH'S IDEA:

We make use of P and P#, where # is larger than any other character.

We search for:

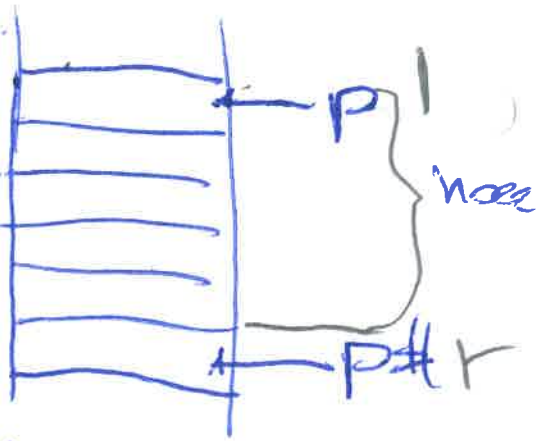
- STARTING POSITION (l) of PREFIX ARRAY A[1,r] through BINARY SEARCH of P in D.
- ENDING POSITION (r) of PREFIX ARRAY A[1,r] through BINARY SEARCH of P# in D.

TWO TYPES of OPERATIONS are supported.

(1) LISTING:

We need to perform:

- 2 BINARY SEARCHES for P and P# $\Rightarrow$ TIME $= O(p \cdot \log n)$
- LISTING of occ, to print the strings between l and r-1 $\Rightarrow O(occ)$ $\Rightarrow$ We may have CACHE MISS per string emitted because of POINTER INDIRECTION.



$$\text{TIME} = O(p \cdot \log n + occ)$$

$$\text{I/Os} = O\left(\frac{p}{B} \cdot \log n + occ\right)$$

NON-CONTIGUOUS LISTING of strings

(2) COUNTING:

We simply need to perform:

- 2 BINARY SEARCHES
- CONSTANT # OPERATIONS to find: #STRINGS prefixed by $P = r - l$.

TIME = $O(P \cdot \log n)$

I/Os = $O(\frac{P}{B} \cdot \log n)$

SPACE of (1) and (2): $O(N)$ ← TOTAL # POINTERS.

NO OPTIMALITY!
↳ Poor scaling!

2 – CONTIGUOUS ALLOCATION of STRINGS.

NEW IDEA: Store strings sorted lexicographically and now also contiguously on disk.

↳ NOW: Pointers' contiguity reflect strings' lexicographic contiguity.

ADVANTAGES:

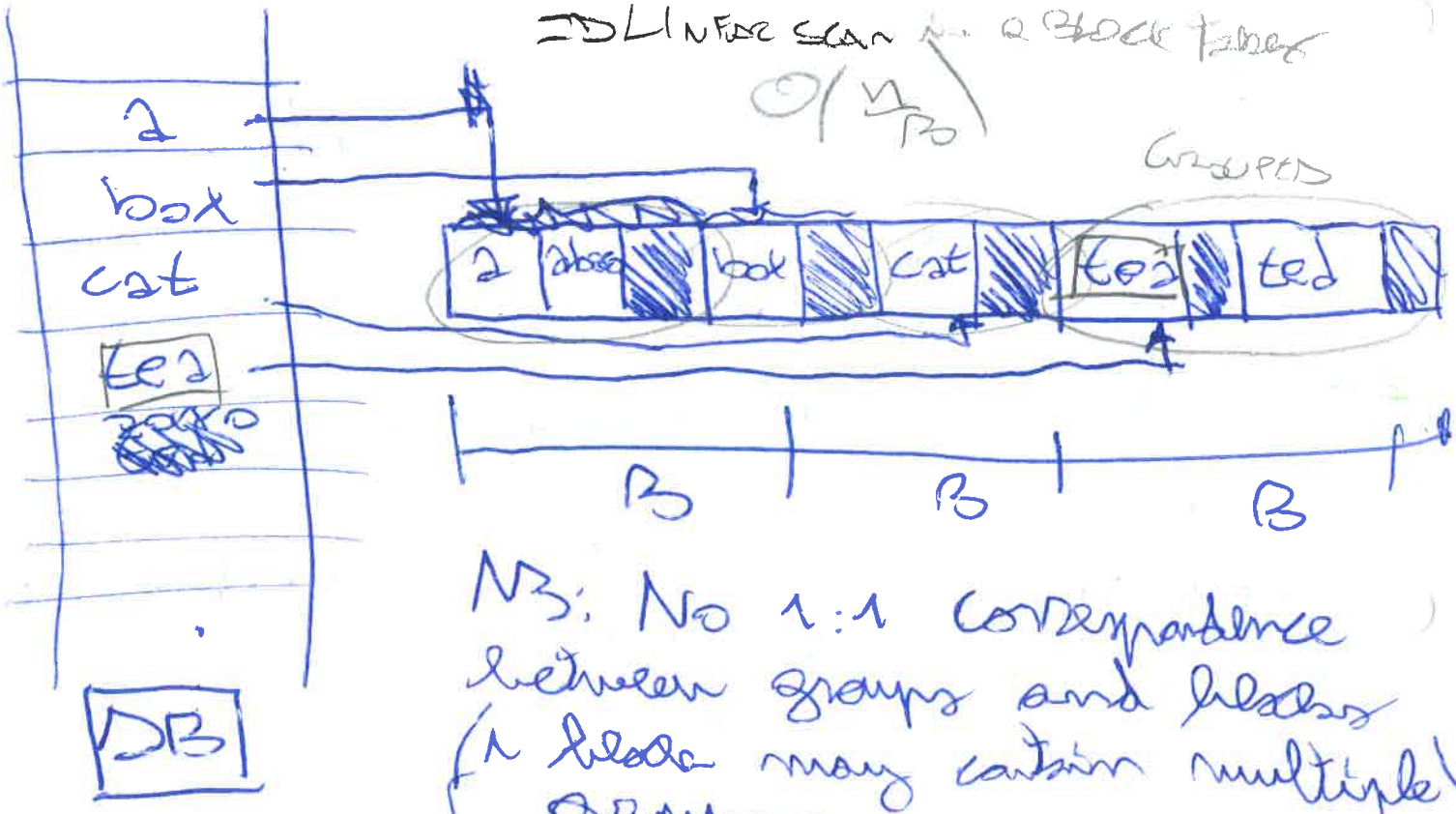
(1) When binary search is performed over strings, it would benefit from a SPEEDUP, as we would perform very close jumps in memory [very few I/Os]

(b) We can apply some compression to contiguous strings, as they now share a PREFIX.

⇓

Because of observation (a), we notice that:

⇒ We are able to group CONTIGUOUS STRINGS into BLOCKS. In the DICTIONARY we just store a POINTER to the first string in the block.



NB: No 1:1 correspondence between groups and blocks (A block may contain multiple groups)

⇒ We are hence able to deploy a 2-LEVEL SEARCH, which operates as follows in stages. 1 & 2 for P.

(1) Perform a BINARY SEARCH of P over DB to identify the lexicographic position of P (i.e: in which group it lies in).

STAGE 2

(2) Use the identified lexicographic position of P to find the block of strings where p lies in.

(3) Linear scan of the strings in the BLOCK to find either l or r.

NB: We repeat these (3) steps for both P and P#.

~~Again, we can perform either:~~

~~- LISTING:~~

I/O cost for each lexicographic position:

$$= \underbrace{\log_2 N}_{\text{BINARY SEARCH cost among blocks}} - \underbrace{\log_2 B}_{\text{over all pointers of Blocks}} = \log_2\left(\frac{N}{B}\right)$$

We can perform either:

- LISTING:

$$\#I/Os = O\left(\frac{P}{B} \cdot \log_2 \frac{N}{B} + \underbrace{\frac{N_{occ}}{B}}_{\text{LIST all strings in contiguous block of text}}\right)$$

- COUNTING

$$\#I/Os = O\left(\frac{P}{B} \cdot \log_2 \frac{N}{B}\right)$$

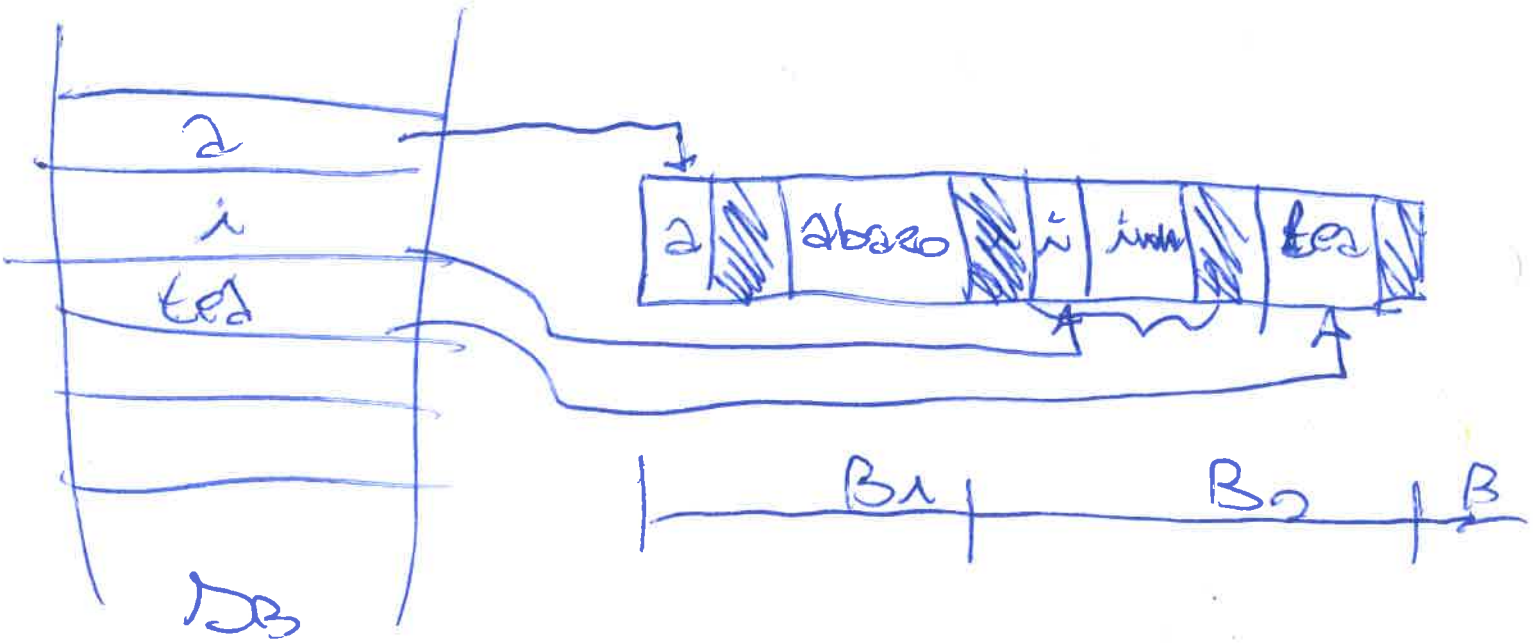
SPACE OCCUPANCY:

$$N + t(1+\varepsilon) \cdot n_B$$

⇒ If strings are very long ($\geq B$), this technique is not advantageous.

⇒ If strings are short ($\leq B$), this technique is advantageous.

PUNCTUATING - VISUALIZED.



P = "in" ⇒ LISTING.

(1) Binary Search of P in DB
⇒ We get "i".

(2) We identify block of strings linked by (i)

(3) Linear scan of block to print "ino"

FRONT-CODING COMPRESSION (STANDARD F2)

Improves on the CONTIGUOUS ALLOCATION of strings (2), lowering the space and time complexity further.

2 EXPLOIT: Contiguous allocation of lexicographically sorted strings → There will likely share a PREFIX!

IDEA: We encode a string with as many possible characters from its preceding string (i.e: LCP between strings S_{i-1} and S_i)

↳ From gli Δ-compression algorithm.

Δ = <alcatraz>, <alcool>, <alcune>, <anelido>, <bo

<0, alcatraz>, <3, ool>, <3, une>, <1, nelido>

↳ |= LCP(S_{i-1}, S_i)

↳ Remaining characters. (Cosa che rappresenta via LCP(S_{i-1}, S_i)||

ISSUE: There is a dependency of a string's decoding on the preceding strings.

↳ WORST-CASE: To decode a string, we may need to go through all the preceding strings.

Ex D = <a>, <aa>, <aaa>, <aaaa>, <aaaaa>

<0,a>, <1,a>, <2,a>, <3,a>, <4,a>

⇒ WORST-CASE DECOMPRESSION:
↳ to decode <aaaaaa>, we need to go through all |D| strings.

~~SOLUTION: FRONT-~~

⇒ **BEST-CASE:** String S_i has $LCP(S_{i-1}, S_i) = 0$ ⇒ i.e.: it is uncompressed ⇒ decoding it costs $O(|S_i|)$.

SOLUTION:
↳ **FRONT-CODING WITH BUCKETING.**

NEW IDEA: We restart front-coding at the beginning of each block, where the **first string of each block** is left **UNCOMPRESSED**.
↳ We **compress each block individually** and store the string length as well, to detect each string separately.

EXAMPLE:

S = <alcatraz>, <alcool> | <alcyone>, <anacleto>

(UNCOMPRESSED, block start: alcatraz; UNCOMPRESSED: alcyone)

B1 | B2

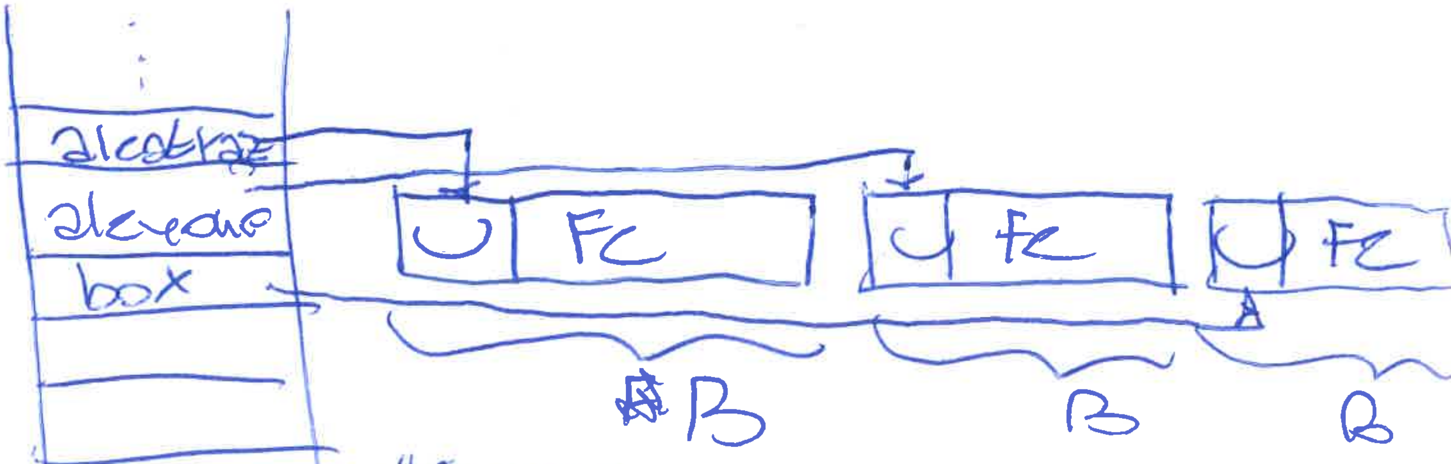
<8, 0, alcatraz> <6, 3, ool> | <7, 0, alcyone>, <8, 1, nacleto>

↳ STRING LENGTH (8)
↳ $l = LCP(S_{i-1}, S_i)$ (0)

⇒ IDEA: Use the UNCOMPRESSED strings (first in the block) to allow retrieval

THE BLOCK. WORST-CASE: Need to decompress all the preceding strings in block's beginning

NEW SITUATION – VISUALIZED:



D_B ← uncompressed

2 PHASES are now:

PHASE 1) We perform BINARY SEARCH just over the UNCOMPRESSED STRINGS to identify the block

PHASE 2) As each block is compressed individually, no extra I/O other than the block itself to read.

(STORED)

SAMPLED STRINGS goes from $\frac{D}{B}$ (NO FC within BLOCKS)

NB: TOTAL SPACE still $FC_B(D)$

$\frac{D}{B}$ → $\frac{FC_B(D)}{B}$

* $FC(B)$ = SPACE required by FC to store all the dictionary strings.

⇒ For the INDEXING & storage ... UNCOMPRESSED

#I/O for PREFIX SEARCH = $O\left(\frac{P}{B} \cdot \log \frac{FC_B(D)}{B}\right)$ (INDEXING)
with FC-with BUCKETING

STILL, we can still have some "PATHOLOGICAL" WORST CASE requiring $O(B^2)$ block accesses.

AVG. # BLOCKS needed = $O(1)$

WORST-CASE # BLOCKS needed = $O(B^2)$

KEYS = <2>, <22>, <222>, <2222>, ...
<0,2> <1,2> <2,2> <3,2>
#LCP = 0, 1, 2, , K-1, K
String

$\Rightarrow K, K-1, K-2, \ldots, 3, 2, 1 = \Theta(K^2)$

Because: $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$

GAUSS

#WORST-CASE # SHARED CHARACTERS = K^2

LISTING:

#I/Os = $O\left(\frac{P}{B} \cdot \log \frac{FC_B(D)}{B} + \frac{FC_B(D)}{B}\right)$

LISTING in a BLOCK

COUNTING:

#I/Os = $O\left(\frac{P}{B} \cdot \log \frac{FC_B(D)}{B}\right)$

LOCALITY-PRESERVING FRONT CODING

Elegant variation of front-coding, providing a trade-off between SPACE OCCUPANCY and TIME to decode a STRING.

→ OPTIMAL wrt. #I/Os.

IDEA: A string S is front-coded only if its decoding time is proportional to its length $|S|$. Otherwise, it is written UNCOMPRESSED.

→ Let ε: we compress a string S only if its decoding is OPTIMAL, otherwise we write it uncompressed (i.e. we copy it).

ALGORITHM: Δ#STEPS needed to decode the string

If $df(S) < \varepsilon |S|$ (SYSTEM PARAMETER)
  copy S // WRITE UNCOMPRESSED
else:
  $FC(S)$ ← MAX. # of many steps as $\varepsilon \cdot |S|$

OPTIMAL DECOMPRESSION. $\forall S \in D$.

#I/Os in DECOMPRESSION = $O\left(\frac{|S|}{B \cdot \varepsilon}\right)$

TOTAL SPACE = $O(1+\varepsilon) \cdot FC(D)$ ⇒ SPACE of compressed string is also proportional to ε.

$\varepsilon = \frac{2}{c-2}$

DECOMPRESSING have costs:

- by S is UNCOMPRESSED:

$$\#I/Os = O\left(\frac{|S|}{B}\right)$$

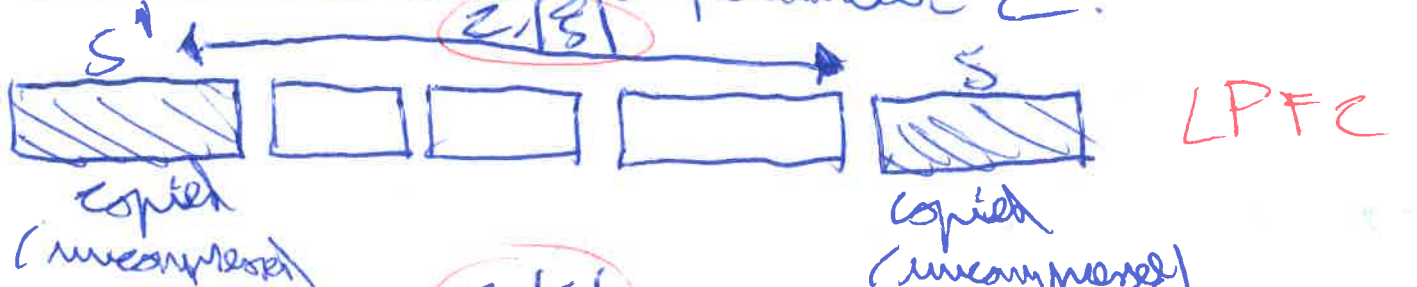
- by S is FRONT-CODED:

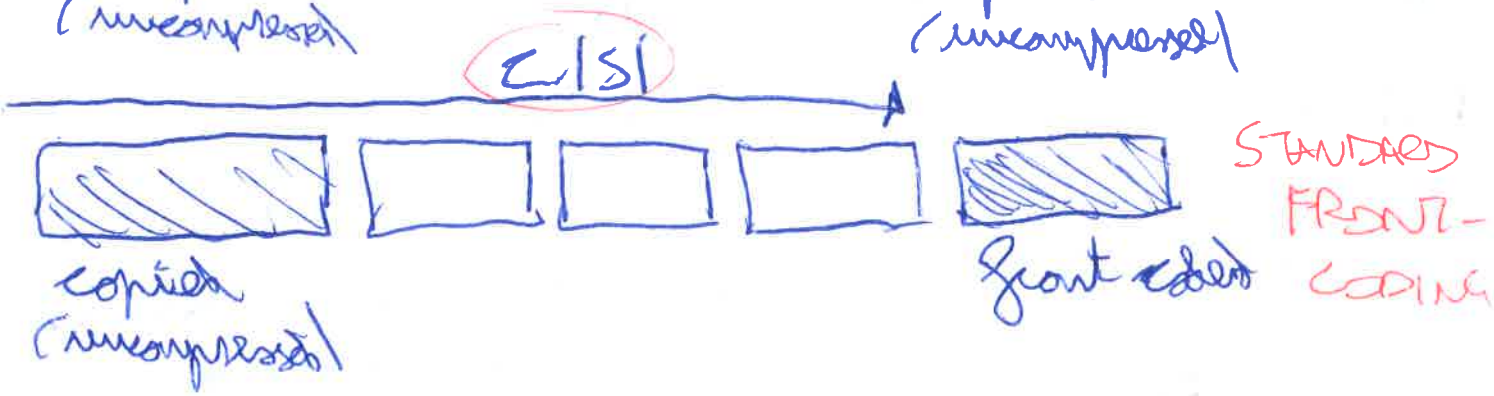
$$\#I/Os = O\left(\frac{C \cdot |S|}{B}\right)$$

PROOF: Consider two cases for copied strings:

- COPIED: #FC-CHARACTERS $\leq \frac{C \cdot |S|}{2}$

We want to show that strings left uncompressed by LPFC (and were instead compressed by the classic front coding), have a length controllable through parameter C.





CLASS EXERCISE – FRONT-CODING

S = {aabb, aaabbb, abbbc, abbbd, abbbe, abc}

FC – COMPRESSION (standard)

<0, aabb>; <2, abbbb>; <1, bbbc>; <4, d>; <4, e>; <2, c>

LPFC with c = 1

Recall the compression algorithm:

if D(S) > c·|S| // #STEPS you are allowed to go back for

  copy S

else:

  FC(S) // FIND ALLO (CONTANDO ANCHE I NUMERI) // CONTARE SOLO CARATTERI DELLE STRINGHE PRECEDENTI

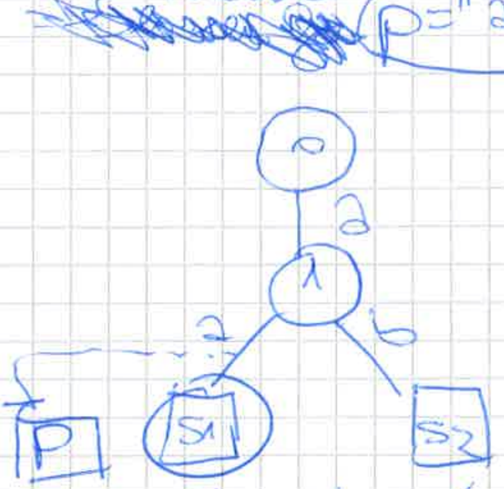
<0, aabb>; <2, abbbb>; <0, abbbc>; <0, abbbd>; <0, abbbe>

LINEAR SCANS TO chars (OR NUMBERS)

→ Build PATRICIA TRIE on two strings

S1 = "aabb" S2 = "abbbd"

insert p = "aaabba"



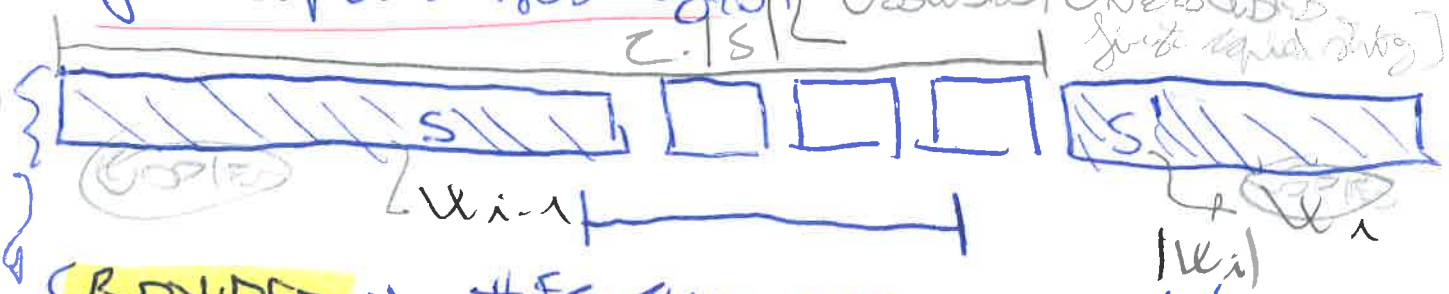
aabb aaabba

(1) S1 is reached. LCP(aabb, aaabba) = aa

⇒ ℓ = 3

PROOF: That the TOTAL #CHARACTERS in an UNCROWDED / CROWDED string is proportional to the length of its first copied string (UNCROWDED one)

⇒ We consider two cases of copied strings, based on the #FC-CHARS that lie between two consecutive occurrences of a copied string (s)



CROWDED STRING ⇒ #FC-CHARACTERS (following / before it) $\leq \frac{c.|s|}{2}$

UNCROWDED STRING ⇒ #FC-CHARACTERS $\geq \frac{c.|s|}{2}$

⇒ We would like to BOUND the total length of copied strings by partitioning copied strings into CHUNKS.

CHUNK = $[w_1 | w_2 | \dots | w_x]$

UNCROWDED COPIED STRING, CROWDED COPIED STRINGS

Take any CROWDED w_i: We have $|w_{i-1}| \geq \frac{c.|w_i|}{2}$ → CROWDED

UNCROWDED

$$|e_{i-1}| \geq \frac{c \cdot |e_i|}{2}$$

$$\Rightarrow |e_i| \leq \frac{2}{c} \cdot |e_{i-1}|$$

Let's consider the TOTAL LENGTH of a CHAIN:

$$|e_1| + |e_2| + \dots + |e_x|$$

$$\sum_{i=1}^{x} |e_i| = |e_1| + \sum_{i=2}^{x} |e_i| \leq$$

$$\Rightarrow \sum_{i=1}^{x} |e_i| \leq |e_1| + \sum_{i=2}^{x} \left(\frac{2}{c}\right) |e_{i-1}|$$

We can observe that:

$$|e_i| < \frac{2}{c} |e_{i-1}| < \left(\frac{2}{c}\right)^2 |e_{i-2}| < \left(\frac{2}{c}\right)^3 |e_{i-3}| \dots$$

And therefore:

$$|e_i| < \left(\frac{2}{c}\right)^{i-1} |e_1|$$

And so, considering the TOTAL LENGTH:

$$\sum_{i=1}^{x} |e_i| \leq |e_1| + \sum_{i=2}^{x} \left(\frac{2}{c}\right)^{i-1} |e_1|$$

$$\left(\frac{2}{c}\right)^0 |e_1| + \sum_{i=2}^{x} \left(\frac{2}{c}\right)^{i-1} |e_1|$$

$$\left(\frac{2}{c}\right)^0 |e_1| + \sum_{i=1}^{x} \left(\frac{2}{c}\right)^{i} |e_1|$$

$$\sum_{i=0}^{x} \left(\frac{2}{c}\right)^{i} \cdot |e_1|$$

$$\Rightarrow \sum_{i=1}^{\infty} |v_i| \leq \sum_{i=0}^{\infty} \left(\frac{2}{c}\right)^i |v_1|$$

$$\frac{\lambda}{\lambda - 2} = \frac{1}{1 - \frac{2}{c}} = \frac{c}{c-2}$$

$$\sum_{i=1}^{\infty} |v_i| \leq |v_1| \cdot \frac{2c}{c-2}$$

$FC(v_1)$

We have hence shown that every chain is bounded by $|v_1|$ and a constant!

$\Rightarrow$ Since v_1 is UNBOUNDED, it will be preceded by at least $\frac{c \cdot |v_1|}{2}$ FC-characters

$v_1 \Rightarrow$ UNBOUNDED v_1

i.e.: we known that!

#FC-characters before v_1 $\geq \frac{c \cdot |v_1|}{2}$ $\frac{c}{2} FC(D)$

And the #FC-characters before v_1 is bounded by $FC(D)$.

$FC(D) \Rightarrow FC(v_1) \geq \frac{c \cdot |v_1|}{2}$

$\Rightarrow$ The length of the traversed string v_1 is hence:

$$|v_1| \leq \frac{2}{c} FC(v_1)$$

$|v_1| \leq \frac{2}{c} FC(v_1)$

We have Gauss test: (UNCOMPRESSED STRING AND BOUNDS)

$$|w_1| < \frac{2}{c} FC(w_i) < \frac{2}{c} FC(D)$$

And test:

$$\sum_{i=1}^{t} |w_i| < |w_1| \cdot \frac{2c}{c-2}$$

PLUGGING IN:

$$\sum_{i=1}^{t} |w_i| < \frac{2}{c} FC(D) \cdot \frac{2c}{c-2}$$

$$\sum_{i=1}^{t} |w_i| < \frac{2}{c-2} FC(D)$$

$$\Rightarrow c = \frac{2}{c-2} \quad \text{Q.E.D.}$$

ADVANTAGES OF LPFC:

Compressed-storage, acting as SPACE BOOSTER!
We are able to uncompress any string at any point in time, and we do not get "PARADOXICAL CASES" of FRACTAL.
→ NO SLOWDOWN on COMPLEXITY!

OPTIMAL I/OS & TIME!

INTERPOLATION SEARCH

IDEA: Combine BINARY SEARCH with bucketing to achieve a very fast search strategy.

INPUT: Sorted array $X[1 \dots m]$, array size

	B1			B3		B6	B7			B10	B11	B12
X =	1	2	3	8	9	17	19	20	28	30	32	36
	1	2	3	4	5	6	7	8	9	10	11	12

BUCKETS' CREATION

X is sub-divided into X_m BUCKETS of size b.

$$b = \left\lceil \frac{X_m - X_1 + 1}{m} \right\rceil = \frac{X_{MAX} - X_{MIN} + 1}{m}$$

X_m: last (and largest) element

because sorted array

EXAMPLE:

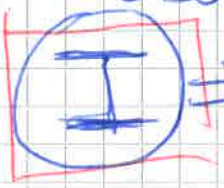
$$b = \frac{36 - 1 + 1}{12} = 3 \Rightarrow \text{BUCKET SIZE}$$

$$\#\text{BUCKETS} = \frac{X_m}{b} = \frac{36}{3} = 12$$

BUCKETS: [1-3] [4-6] [7-9] [10-12] [13-15] [16-18] [19-21] [22-24] [25-27] [28-30] [31-33] [34-36]

B1 B2 B3 B4 B5 B6 B7 B8 B9 B10

Now also consider an array I, containing the starting and ending position of every bucket.



⇒ Used to learn how to jump to a certain bucket in $O(1)$ time.

I → Starting & ending Pos. of every bucket (k)

PSEUDOCODE:

Int. Search(y)

① $j = \left\lfloor \frac{y - x_1}{b} \right\rfloor + 1$ // j-th BUCKET for y

② BINARY SEARCH for y in the portion of X delimited by I[j]

① Find Bucket B_j

② Binary search for y over B_j.

TIME: $O($ FIND BUCKET B_j + BINARY SEARCH over B_j $)$

(PRELIMINARY) 2 Results ESTIMATE

$O(1 + \log_2 b) = O(\log_2 b)$

↳ Pretty inaccurate estimate of the TIME COMPLEXITY

SPACE: $O(m)$

↳ We need to store the pointers & elements therein

EXERCISE (IN CLASS)

INTERPOLATION SEARCH EXERCISE.

1, 3, 4, 9, 10, 13, 20, 21, 28, 30, 32, 36

(1) PARTITION items into BUCKETS, where the BUCKET SIZE IS:

$$b = \frac{X_m - X_1 + 1}{m} = \frac{36 - 1 + 1}{12} = \frac{36}{12} = 3$$

$$\#\text{BUCKETS} = \frac{X_m}{b} = \frac{36}{3} = 12, \text{ with 3 elements each}$$

BUCKETS' REPRESENTATION — 2 WAYS:

(1) <Starting POSITION (element), bucket length>

(2) <Starting POSITION (element), listing position>

⇒ We go for REPRESENTATION (1).

(2) Fill-in values into the corresponding bucket.

1	1,3	4	4	7	9	10	10	13	∅		
16	19	19	20,21	22	28	25	∅				
28	28,30	31	32	34	36						

(3) Look for ITEM (21) (for instance).

⇒ We retrieve bucket (19), then perform a BINARY SEARCH into this bucket.

over [20, 21]

↳ O(n) time!

(21) is then retrieved!

GOAL: We would like to better estimate the runtime of INTERPOLATION SEARCH.

TIME for INT. SEARCH $= O(1 + \log_2 b) = O(\log_2 b)$

- 1: Find BUCKET by
- $\log_2 b$: BINARY SEARCH over bucket by.

THEOREM: (INTERPOLATION)
Binary search cost ~~is~~ in dictionary of size n DOMINATES the cost.

$O(\log \Delta)$ in the worst-case, where.

$$\Delta = \frac{\text{MAX. GAP between } X_i \text{ and } X_{i-1}}{\text{MIN. GAP between } X_i \text{ and } X_{i-1}}$$

EXTRA SPACE $= O(m)$. (OPTIMAL ASYMPTOTICALLY)

PROOF:
CORRECTNESS is immediate.
TIME COMPLEXITY: The maximum of a series of integers is at least as large as their average (GAP)

- MAX. GAP $\geq$ AVG. GAP $\Rightarrow$ LOWER BOUND

$$\max_{i=2,\dots,m} (x_i - x_{i-1}) \geq \frac{1}{m} \sum_{i=2}^{m} x_i - x_{i-1} =$$

MAX. GAP $\geq$ AVG. GAP

$$\max_{i=2,\dots,m} x_i - x_{i-1} \geq \sum_{i=2}^{m} \frac{x_i - x_{i-1}}{m-1} = \frac{x_m - x_1}{m-1} + 1$$

TELESCOPIC SERIES

$$\Rightarrow \max_{i=2,\dots,m} x_i - x_{i-1} \geq \frac{x_m - x_1 + 1}{m} = b$$

MAX. BIN SIZE

UPPER BOUND

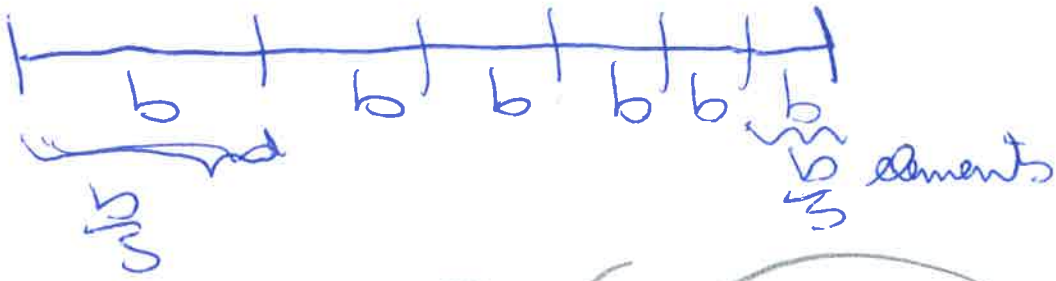
- Also, the max. # integers that can belong to any bin (b) can be upper bounded.

$\Rightarrow$ Integers of X are spaced apart by:

$$S = \min_{i=2,\dots,m} (x_i - x_{i-1})$$

UNITS per BIN.

$\Rightarrow$ Every bin of size b has at most $\frac{b}{S}$ elements.



$$\Rightarrow |B| \leq \left(\frac{b}{S}\right)!$$

MAX. BUCKET SIZE

$\Rightarrow$ Let's put together the LOWER & UPPER BOUND in the definition of Δ.

$$\Delta = \frac{\text{max. GAP between } x_i \text{ and } x_{i-1}}{\text{min. GAP between } x_i \text{ and } x_{i-1}}$$

~~$\Rightarrow$ VB Analysis~~ And we have found that:

$$|B_i| \le b \Rightarrow \frac{\max_{i=2,\dots,m}(x_i - x_{i-1})}{\min_{i=2,\dots,m}(x_i - x_{i-1})} = \Delta$$

$$|B_i| \le b \le \Delta$$

$\Rightarrow$ The ALGORITHM's complexity hence depends on the SPARSENESS of values, not on their MAX/MIN value.

$\Rightarrow$ The more sparse items are, the faster the ALG. runs [fewer items in each single BUCKET]

$\Rightarrow$ DOUBLINGS to Δ.

[Better! more buckets with few values]

WORST-CASE SEARCH TIME:

If whole array X ends up in a single BIN, the WORST-CASE search time is hence:

$$O(\log \min(\Delta, n))$$

[NO WORSE THAN BINARY SEARCH] Same as performance BINARY SEARCH

$\Rightarrow$ We are more interested in the AVERAGE SEARCH TIME.

AVERAGE SEARCH TIME.

The time complexity is $O(\log \log m)$, if the integers are drawn uniformly at random (i.e. uniform DISTRIBUTION) from a set M.

Proof: We want to estimate Δ with high PROBABILITY.

(NB: $\log \log m \ll \log m$ one order of magnitude)

$\Rightarrow$ Assume integers are drawn uniformly from $[0, U-1]$.

We partition the range into r ranges $\Rightarrow$ Each range has $\frac{U}{r}$ elements, where

$$r = \frac{m}{2 \cdot \log m} = \#\text{RANGES}.$$

We have r ranges.

Pr[one integer belongs to a range] $= \frac{1}{r}$
out of r existing ranges

$P\{$ one specific range does not contain one integer $\} = 1 - \frac{1}{r}$

$P\{$ one specific range does not contain any integer $\} = \left(1 - \frac{1}{r}\right)^{\#\text{ of ranges}} = \left(1 - \frac{1}{r}\right)^{m}$

We know test $r = \frac{m}{2 \cdot \log m}$

$\Rightarrow P\{$ one specific range does not contain any integer $\} = \left(1 - \frac{2 \log m}{m}\right)^{m}$

$= \left[\left(1 - \frac{2 \log m}{m}\right)^{\frac{m}{2 \log_2 m}}\right]^{2 \log_2 m}$

$= e^{-2 \log m}$

$\Rightarrow O\left(e^{-2 \log m}\right) = O\left(\frac{1}{m^2}\right)$

$= O\left(m \cdot \frac{1}{m^2}\right) = O\left(\frac{1}{m}\right)$

$\Rightarrow P\{$ at least one range remains empty $\} \leq O\left(\frac{1}{m}\right)$

(or, at least one range contains at least one element)

If this occurs with high probability, then the max. distance between two adjacent integers must be smaller than twice the range's length.

$$\max_i x_i - x_{i-1} \leq 2 \cdot \frac{U}{r} = \frac{2 \cdot U \cdot 2 \cdot \log m}{m}$$

$$r = \frac{m}{2 \cdot \log m}$$

$$\Rightarrow \max_i x_i - x_{i-1} = O\left(\frac{U \cdot \log m}{m}\right)$$

Let's now take $r' = \Theta(m \cdot \log m)$ Ranges.

$\Longrightarrow$

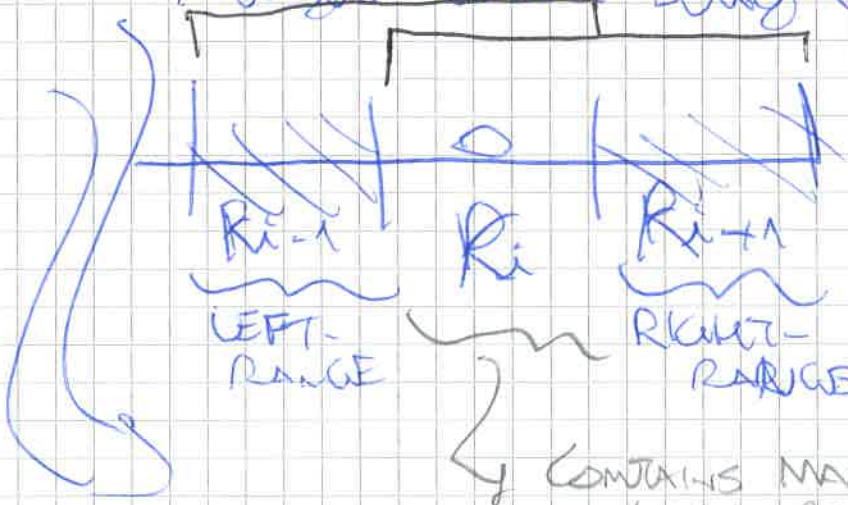
We now want to find

$$\min_i x_i - x_{i-1}$$

⇒ Now I take $r = \Theta(m \cdot \log m)$ ranges.

⇒ (Greater than m itself)

We can prove that every adjacent pair of ranges contains at most one integer with high Probability



R_i = i-th range.

IF a RANGE R_i contains ONE integer, its two adjacent ranges (LEFT & RIGHT) are empty with high Probability.

$$\Rightarrow \min_{i=2}^{m} x_i - x_{i-1} \geq \left\lfloor \frac{U}{r} \right\rfloor = \Theta\left(\frac{U}{m \cdot \log m}\right)$$

Minimum distance between an adjacent pair of integers

Let's now take Δ:

$$\Delta = \frac{\max_i x_i - x_{i-1}}{\min_i x_i - x_{i-1}} \leq \frac{U}{m} \cdot \frac{\log m}{U} = \log m \cdot \log m$$

$$\frac{U}{m \cdot \log m}$$

INTERPOLATION SEARCH'S TIME COMPLEXITY $\Delta = O(\log^2 m)$ WITH HIGH PROB.

#1105: $O\left(\frac{P}{B} \cdot \log \log \frac{N}{B}\right)$

↳ EXP. reduction in search time performance! (wrt. BINARY SEARCH)

NB: If strings are NOT uniformly distributed

↳ We permute them to get a UNIFORM DISTRIBUTION (randomly)

COMPACTED TRIE:

Compacted Trie consist of EFFICIENT data structures for SEARCHING strings.

TRIE → Index complete strings of DB in internal memory

↓

Speed up in first phase of prefix search from $O(\log(\frac{N}{B}))$ to (BINARY SEARCH one) $O(P)$ TIME.

↓

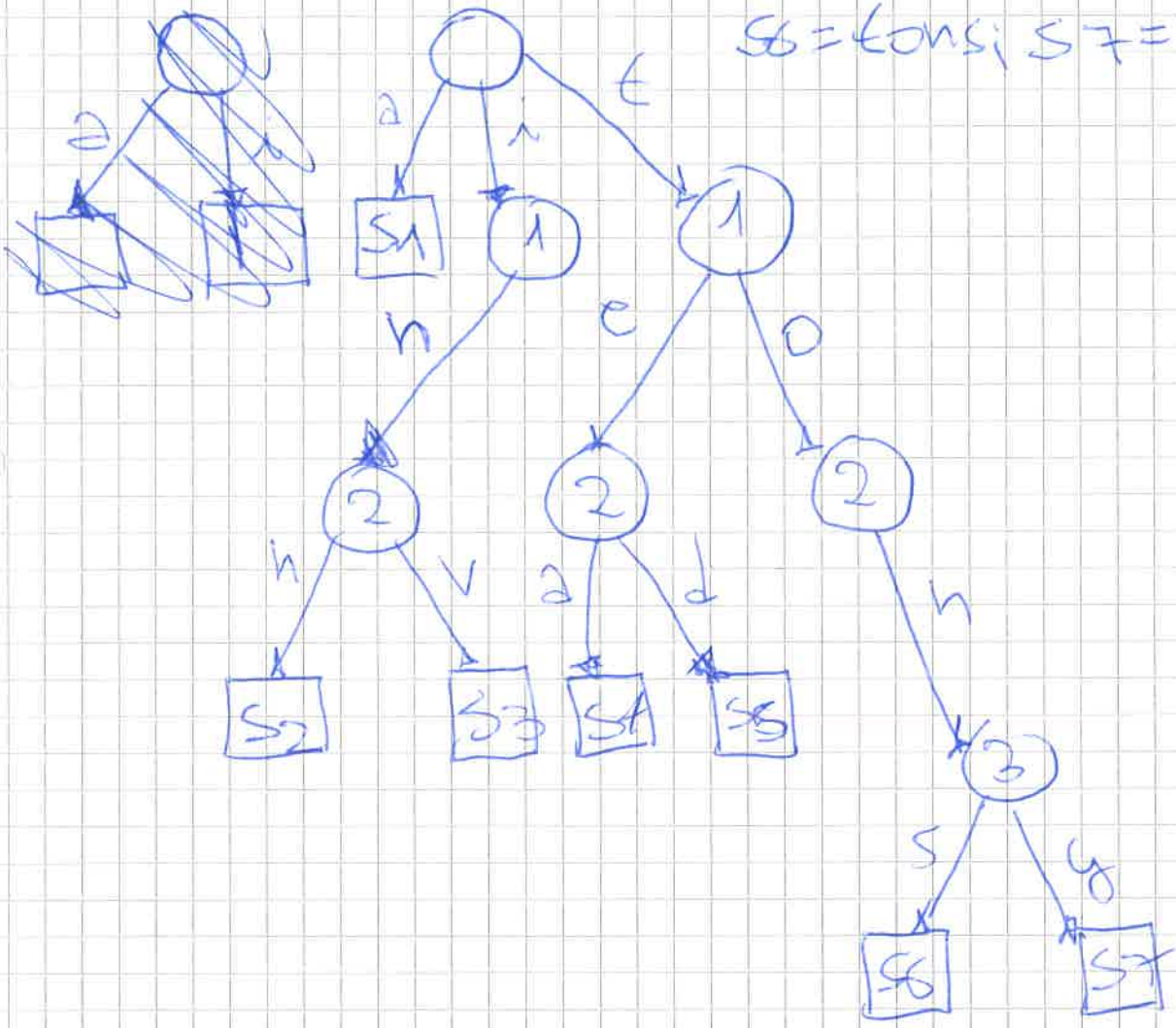
Independent from dictionary size, as in RAM we can manage and address memory cells of $O(\log M)$ bits

+ Also save space! No UNITARY PATHS, as these are compacted!

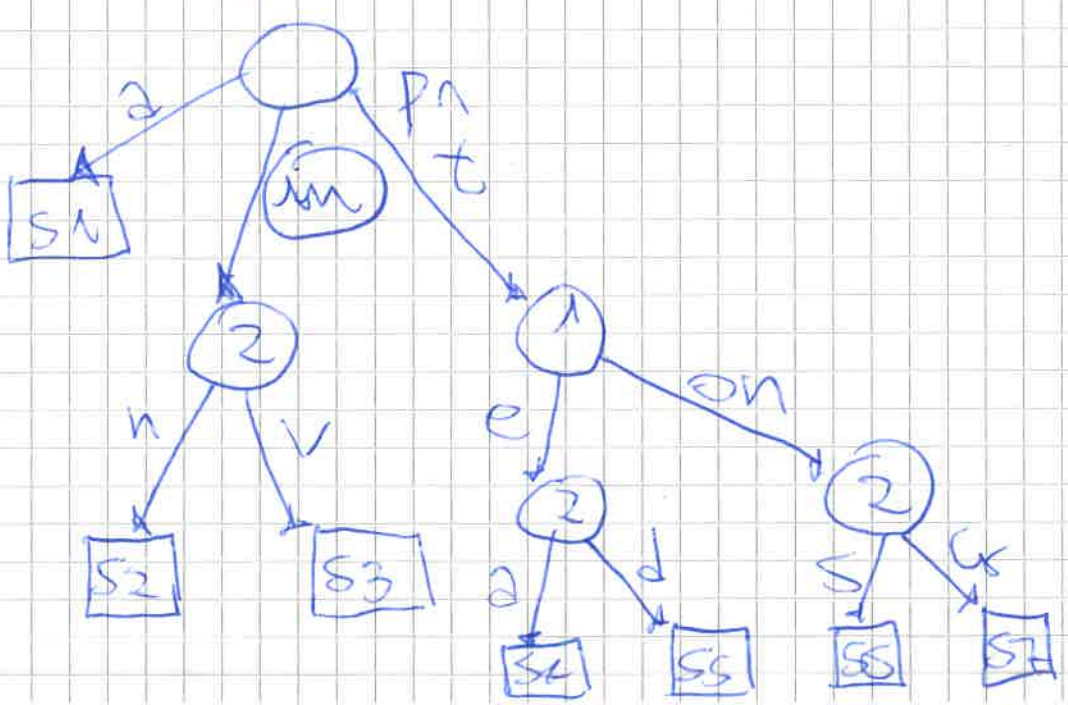
UNCOMPACTED TRIE (Patricia):

Strings:

S_1 = a ; S_2 = inn ; S_3 = inv ;
S_4 = tea ; S_5 = ted ;
S_6 = tons ; S_7 = tony



Corresponding COMPACTED TRIE:



O(p) height

COMPACTED TRIE'S FEATURES

#LEAVES = n

#INTERNAL NODES < n

#NODES + LEAVES < 2n

SPACE = O(n) + O(N) → for edge labels (#NODES = O(n))

PREFIX SEARCH TIME = O(p + nocc)

#I/Os O(p + nocc/B) → #STRINGS prefixed by pattern P.

(if leaves & strings stored contiguously & sorted alphabetically on disk)

EDGES' REPRESENTATION.

Edges are simply represented as pointers to (a part of) characters in leaves' strings.

Ex. P1 "in" = <S2, 1, 2> → length of the substring; 1 → starting index of the substring

Ex2. P2 "on" = <S7, 1, 2> ≈ 12 bytes (4 bytes, 4 bytes, 4 bytes)

Hence, the space required by edges is:

12 bytes · #EDGES = O(n) (proportional to the #STRINGS)

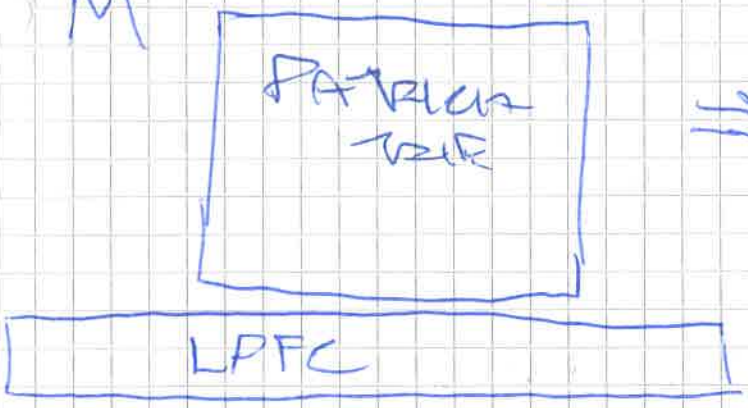
⇒ SPACE is hence proportional to the #STRINGS that can be stored in internal memory (otherwise, linear dependance of #I/Os on pattern length p)

⇓

#STRINGS that can fit into the INDEX.

Ex. 2GB RAM ⇒ ≈ 2.000.000.000 strings can fit in internal memory & be properly indexed.

M



⇒ COMPACTED TRIE stored in M.

ISSUE: What if some strings do not fit in the internal memory M? (i.e.: if N → M)

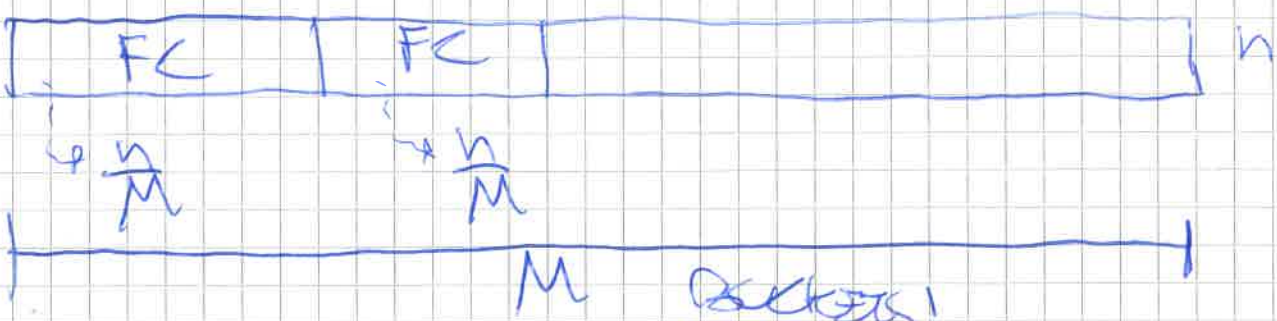
(SAME APPROACH as for COMPACTED TRIES)

#STRINGS also need to be stored on disk, "SAMPLE".

Even more strings ≥ than in M

⇓

We may make use of BUCKETS:



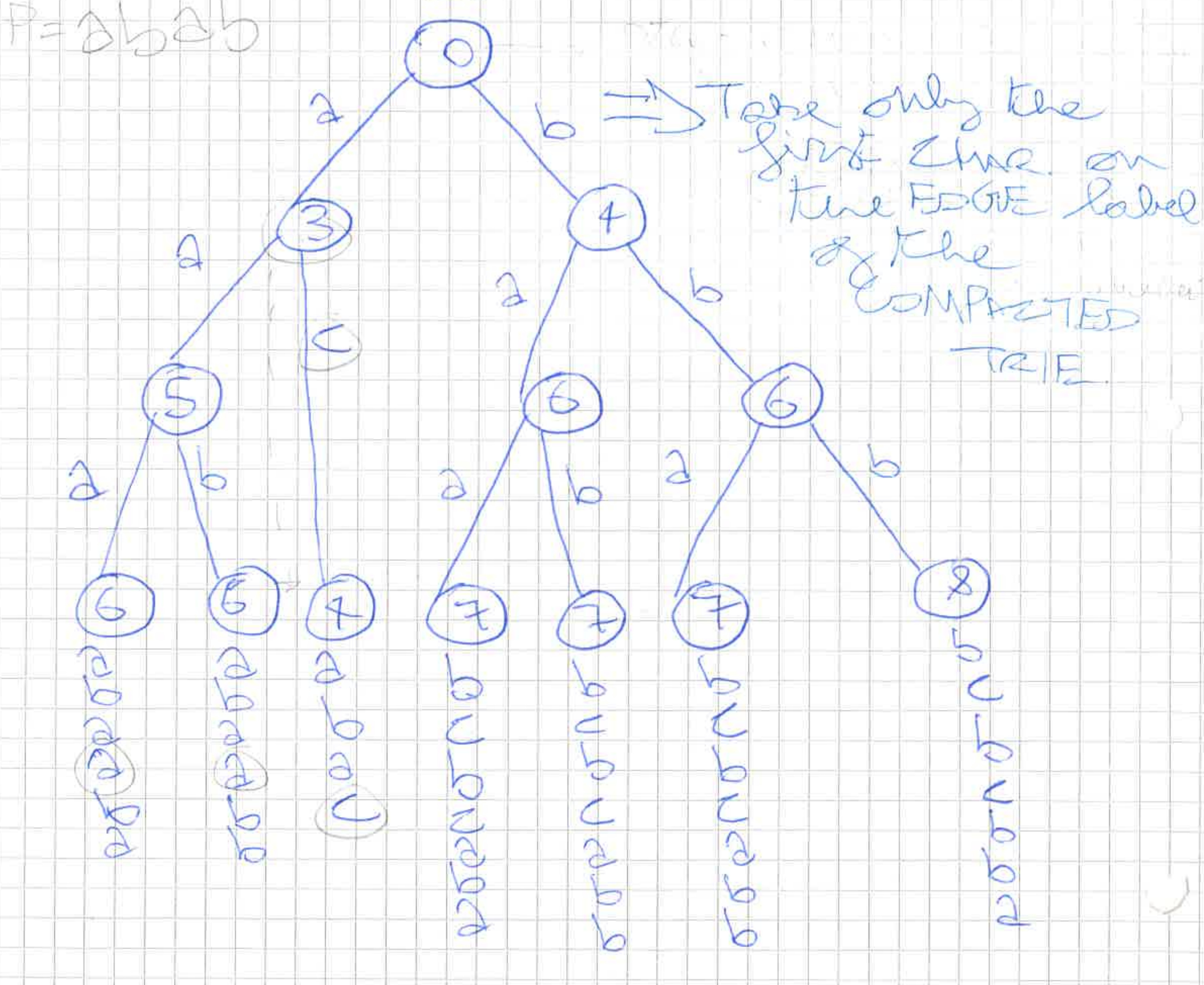
PATRICIA TRIES

Data structure that, after being built, allows for search in O(1) TIME & I/Os of a pattern among a SORTED sequence of strings

ADVANTAGE: Access one single string in O(1) I/O instead of O(P) I/Os

PATRICIA TRIE — COMPACTED TRIE

PATRICIA TRIE:



SEARCH PROCESS IN A PATRICIA TRIE

INPUT: pattern P, patricia TRIE T (pointing to S)

1) In $O(n)$ time, trace path to a leaf, say l, DOWNWARDS until no branching is possible any longer.
[keep going down as long as characters match along the edges]
If a TIE among leaves, pick one randomly.

2) In $O(\frac{P}{B})$ time, compute:

$$L = LCP(P, S)$$

⇒ We will hence know that:
$S[L+1]$ is the MISMATCHING CHARACTER.

$$P[L+1] \neq S[L+1]$$

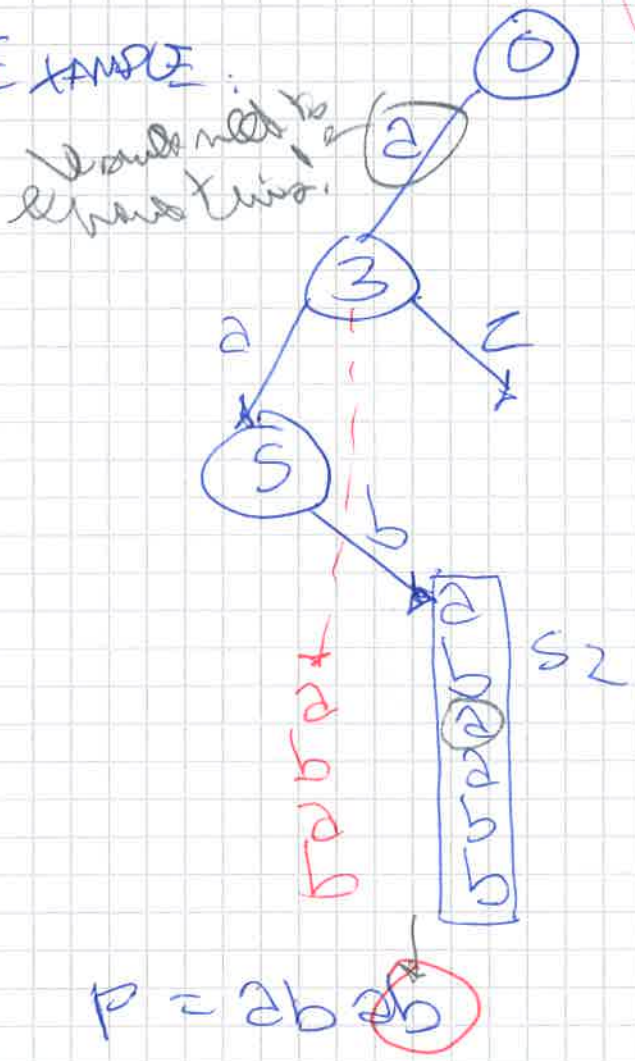
N3: Leaf l has the string having as LCP as P and S.

3) Perform an UPWARDS TRAVERSAL from leaf l up to an EDGE where the mismatching character $S[L+1]$ lies.
(i.e: go up and find $S[L+1]$)

④ Identify the correct branching position for pattern also found.

P = abab

EXAMPLE:



P = abab

① DOWNWARD POSN. Reach S2

② L = LCP(P, S2)

= LCP(abaabb, abab) = 3

③ Compare and find mismatching character at S[L+1] ~~Return~~ ⇒ Increment node ③.

④ Plug in the Pattern correct position.

COSTS IN A PATRICIA TRIE:

The blind search (downward pass) in a PATRICIA TRIE takes $O(p)$ time and $O(\frac{p}{B})$ #I/Os to compare the string identified by blind search.

⇒ Searching for P & P# in the CONTRACTED TRIE, we can find the #OCCURRENCES between P and P#.

SPACE = $O(n) + O(N)$

NODES & EDGES (in heap local) — STRINGS

SEARCH TIME = $O(p)$ (IN MEMORY)

#I/Os for SEARCH = $O(\frac{p}{B})$ ⇒ Because of LCP computation when reaching the leaf

USAGE OF LPFC WITH PATRICIA TRIE.

⇒ If we want to compress strings indexed by the PATRICIA TRIE, we proceed as follows:

- We put in internal memory the PATRICIA TRIE of dictionary D. (#STRINGS $n > M$ ⇒ INTERNAL MEMORY)
- We store on disk the LPFC of the dictionary D' strings.

⇒ In this manner, traversing the PATRICIA TRIE takes $O(p)$ and NO I/Os, as we fully operate in RAM.

⇒ LCP computation takes:

$$O\left(\frac{|S|}{B} + \frac{p}{B}\right) \quad \#I/Os$$

⇒ Needs to decode strings from their LPFC-representation.

~~$O(n)$~~

SPACE: $O(n)$ for PATRICIA TRIE stable in internal memory.

$O(|(1+\varepsilon)|FC(D)||$ on DISK

$$\Rightarrow \#I/Os = O\left(\frac{(1+\varepsilon)\cdot |FC(D)|}{B}\right)$$

If $n = \Omega(M)$ (i.e. strings of the PATRICIA TRIE do not fit in internal memory)

⇒ We need to perform some "BUCKETING" and store a sample of strings in the PATRICIA TRIE.

(Ex.: First string & last string BUCKET stored in PATRICIA TRIE)

EXERCISE (IN CLASS)

a) Build a PATRICIA TRIE for:

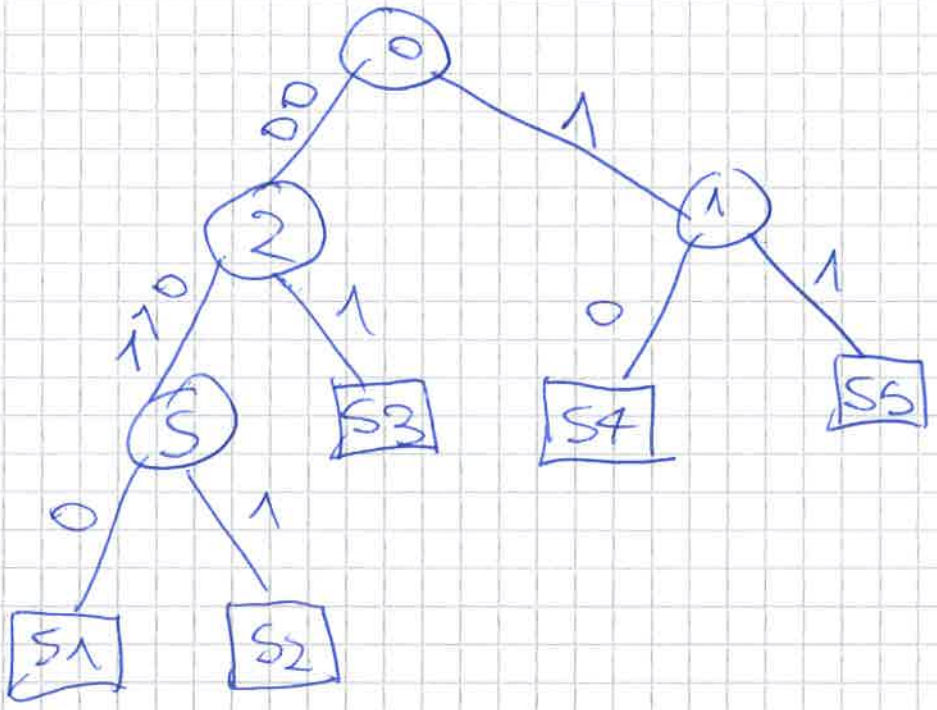
$S = \{000\underline{1100}, 000\underline{1110}, 00\underline{10}, \underline{101}, \underline{111}\}$

It is useful to build first the corresponding COMPACTED TRIE, and then the PATRICIA TRIE.

$S = \{\underbrace{000\underline{1100}}_{S1}, \underbrace{000\underline{1110}}_{S2}, \underbrace{00\underline{10}}_{S3}, \underbrace{\underline{101}}_{S4}, \underbrace{\underline{111}}_{S5}\}$

⇒ NODES have the # common characters
EDGES have the common characters themselves
LEAVES have the STRINGS we want to store in the TRIE.

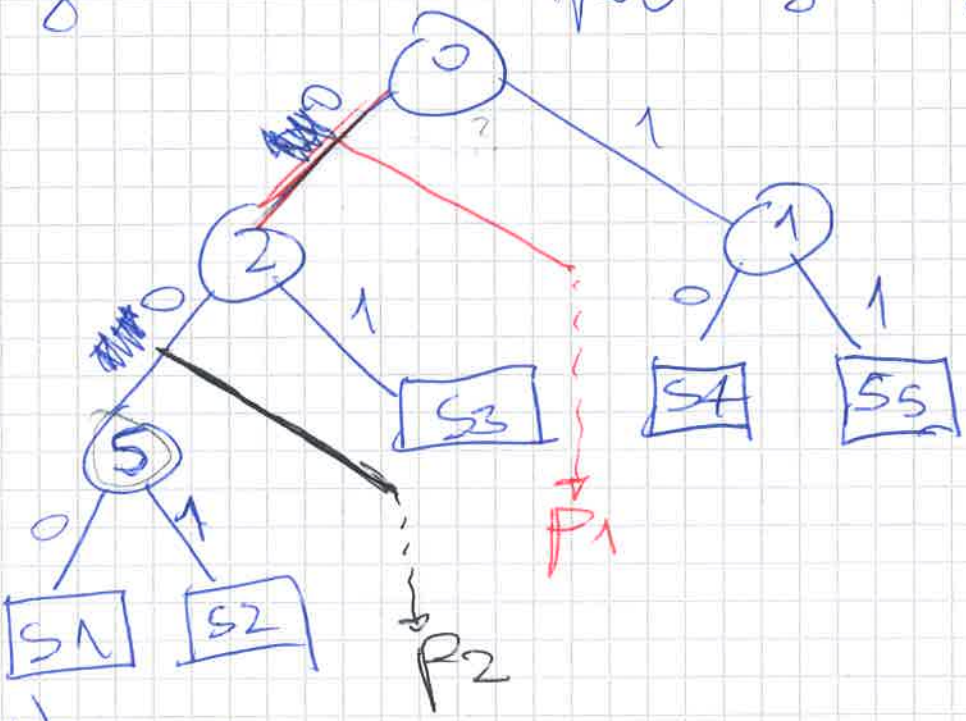
COMPACTED TRIE:



⇒ The corresponding PATRICIA TRIE can be built simply by taking the first character of each edge from the COMPACTED TRIE.

PATRICIA TRIE:

Like the COMPACTED TRIE, but write the first character per edge only.



b) Find the LEXICOGRAPHIC POSITION for:

① $P_1 = 01000$

② $P_2 = 0101$

① $P_1 = 01000$

a) STEPS: 1) DOWNWARD PATH

(Navigate PATRICIA TRIE ~~following~~ as long as you find matching characters)

$\Rightarrow$ S_1 returned

b) $LCP(P_1, S_1) = 1 = L$

$L = LCP(000110, 01000)$

c) Upward traversal up to the mismatching character $P_1[L+1]$

P1[2] ≤ 0?

P1[2] ≤ 0?

L+1

1 ≤ 0? No!

⇒ RIGHT of the edge!

2) P2 = 01001

- DOWNWARD PATH:

We never come to (5) ⇒ We can pick either (S1) or (S2) ⇒ We pick S1

- L = LCP(P2, S1)

= LCP(21001, 00011001) = 1

⇒ Where do we put P2 now?

⇒ Go up to node 2

LCP in LINEAR TIME - KASAI'S ALGORITHM

S = aaaazaak$
1 2 3 4 5 6 7 8 9

$O(n)$ $O(n^2)$

SORTED SUF(T)	SA	LCP	# LINEAR COMPARISONS	BRUTE-FORCE # COMPARISONS
aaaazaak	1	3	4	
aaazaak	2	2	3	
aak	6	2	1	
aazaak	3	1	1	
ak	7	1	1	
azaak	4	0	1	
k	8	0	1	
zaak	5	—	—	

SA^{-1} => Given an SA, it returns its Rank

h = # surely shared chars

SA^{-1}

1	1
2	2
3	4
4	6
5	8
6	3
7	5
8	7

LINEAR COMPARISONS

SA=1 { aaaazaak, aaazaak => # comps = 4, LCP = 3 => h = 3

SA=3 { h = 2; aak, aazaak => # comps = 1

SA=4 { h = 1; ak, azaak => # comps = 1

SA=5 { h = 0; k, zaak; # comps = 1

SA = 6

aaaaaak ⇒ #comps. = 3
aak
h = 2

SA = 7
h = ~~2~~ 1
aaaaaak
ak
#comps = 1

SA = 8
h = ~~1~~ 0
aaaak
k
#comps = 1

⇒ DFS: Process by groups of 2, & check LCP between suff$_i$ and suff$_{i-1}$

(10) SUBSTRING SEARCH.

We are presently interested in solving the FULL-TEXT SEARCHING or SUBSTRING SEARCHING problem.

INPUT: Text string $T[1,n]$

We want to:

- RETRIEVE (or count) all text positions where a query pattern $P[1,p]$ occurs as a substring of T.

Ex: $T = \underset{123456}{trento}$

$P = to$

output: $T[5,6]$ or (1)

"# occurrences of P in T.

SUFFIX - DEFINITION:

Consider a text T ⇒ No suffix is a PREFIX of another suffix (obbligato $)

$SUFF_i = T[i,n]$ (i.e.: Suff$_i$ occurs at positions i,n)

[The opposite of a PREFIX] ⇒ occurs from 1 to n.

Ex: $T = \underset{1234567}{missis}\underset{89\,10\,11}{sippi\$}$

Suff$_6$ = sippi$

PREFIX & SUFFIX:

↳ $P = T[\lambda, \lambda + p - \lambda]$

↳ Pattern $P[\lambda, p]$ is a PREFIX & [redacted]

$SUFF_\lambda = T[i, n]$

$T =$ "mississippi"
1 2 3 4

$P =$ "siss"

⇒ P occurs at pos. 4 and it matches the string $T[4, \lambda 2]$

$SUFF_4$.

BRUTE-FORCE APPROACH FOR SUBSTRING SEARCH:

⇒ Compare P against all possible sub-strings of $T[\lambda, n]$

↳ WORST-CASE $O(n \cdot p)$

$T =$ mississippi #
1 2 3 4 5 6 7 8 9 λ0 λ1 λ2

$P =$ "iss"

⇒ n comparisons, taking p each one.

① mississippi ; "iss"
② ississippi ; "iss"
③ ssissippi ; "iss"
④ sissippi ; "iss"
⑤ issippi ; "iss"
⋮
⑫ ⇒ "ississippi" and "issippi" are returned.

SMARTER APPROACH - DATA STRUCTURES.

We consider some data structures to quickly retrieve & efficiently store SUBSTRINGS (i.e. SUFFIXES)

4. SUFFIX ARRAY (SA[1..n])

It is an array of pointers to all text suffixes, ordered LEXICOGRAPHICALLY.

SA[i] = i-th smallest text index.

$SUFF_{SA[1]} < SUFF_{SA[2]} < \dots < SUFF_{SA[n]}$

↑ Sorting by LEXICOGRAPHICAL ORDER.

SPACE (BITS) $= O(n \cdot \log n) + O(n \cdot \log \Phi)$

SPACE in BITS of pointers to elements of SA

POINTERS to SORTED SUFFIXES.

Ex: T = mississippi$

1 2 3 4 5 6 7 8 9 10 11 12

SORTED SUFFIX	SA (index of SUFFIX)	LCP	BRUTE FORCE: LCP(suff SA[i], suff SA[i+1])
$	12	0	1
i$	11	1	2
ippi$	8	1	2
issippi$	5	0	1
mississippi$	1	0	1
pi$	10	1	2
ppi$	9	0	1
sippi$	7	2	3
sissippi$	4	1	2
ssippi$	3	3	4
ssissippi$	2		1

SUBSTRING-SEARCH PROBLEM:

"Searching for a pattern P as a substring of T = searching for pattern P as a PREFIX of a string in SA(T)."

(We have re-phrased our problem, after looking at SA(T)).

⇒ To solve this problem, we have to note 2 OBSERVATIONS:

1) SUFFIXES prefixed by P occur contiguously in SA(T)

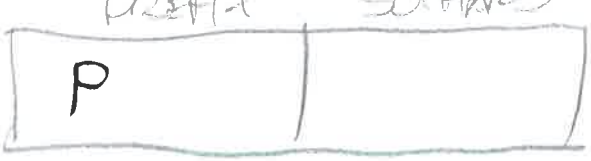
Ex: P = "si"

sippi$
sissippi$ } CONTIGUOUS STRINGS (with same prefix)
ssippi$

2) The lexicographic position of P in SA(T) immediately precedes the block of suffixes prefixed by P.

Ex: P = "si"

si
sippi$
sissippi$
ssippi$



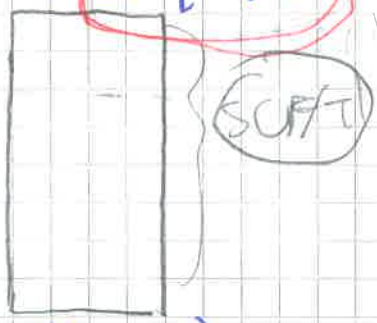
⇒ These properties help in deploying "PREFIX SEARCH" to achieve SUBSTRING SEARCH.

LCP = Longest Common Prefix among two consecutive suffixes:

SUFF SA[i] and SUFF SA[i+1]

BINARY SEARCH FOR SUB-STRING-SEARCH PROBLEM:

We have reduced the problem of finding a SUB-STRING to finding a PREFIX ⇒ Hence, we can make use of the STRING-BASED binary search AL over SA(T)




```
SUBSTRING SEARCH(P, SA(T))
    L=1, R=n;
    while(L≠R) do
        M = ⌊(L+R)/2⌋
        if (strncmp(P, SUFFM, p) > 0)
            L = M+1          // Must need to go right of M
        else
            R = M,           // need to go left of M
        endif
    endwhile
    if(strncmp(P, SUFFL, p)==0)
        return L;
    else
        return -1;
    endif
```

The only difference lies in WHERE we apply the ALG. Here, we are applying it over a SUFFIX ARRAY ⇒ Go UP/DOWN along it.

BINARY SEARCH over SA ⇒ $O(\log n)$ = #COMPARISONS

Scanning of a STRING ⇒ $O(p)$ (WORST CASE)
(by applying strcmp)

⇒ TOTAL TIME = $O(p \cdot \log n)$
(SEARCH + COMPARISONS)

~~If we also wanted to COUNT #OCCURRENCES~~

#I/Os: $O\left(\frac{p}{B} \log n\right)$

strcmp ⇒ Scanning of string required
#I/Os = $O\left(\frac{p}{B}\right)$

To count the #OCCURRENCES. $O(occ)$
⇒ $O\left(\frac{occ}{B}\right)$ = #I/Os.

SPACE = ~~$n \log n$~~ $n \cdot \log n + n \cdot \log \Theta$

SPACE = $n \cdot (\underbrace{\log n}_{SA(T)} + \underbrace{\log \Theta}_{TEXT_i})$ BITS

APPLICATIONS of SUB-STRING SEARCH:
Bio-informatics / ~~Google~~ DNA sequencing

EXAMPLE – SUBSTRING SEARCH ALGORITHM
⇒ BINARY-SEARCH-LIKE APPROACH

We have text: p = "ssi"

	SORTED SUF(T):	SORTED SUF(T)	SORTED SUF(T)
1	$		
2	i$		
3	ippi$	⇒	
4	issippi$		
5	ississippi$		
6	mississippi$		
7	pi$	pi$	
8	ppi$	ppi$	
9	sippi$	sippi$	
10	sissippi$	sissippi$ mississippi$	sissippi$
11	ssippi$	ssippi$	ssippi$
12	ssissippi$	ssissippi$	ssissippi$

① strncmp(ssi, mississippi) = 1 ⇒
strncmp(ssi, sissippi) = 1
strncmp(ssi, ssippi) = 1 ⇒ set l = m+1
⇒ l = r

⇒ Resulting Interval:
r, l → ssippi$

Nb. RLCP $\in$
LLRP could be computed in
$O(n)$ TIME!

ALGORITHM'S PROPERTY (to obtain ben better

COMPLEXITY)

→ The comparison between P and suff_M does NOT need to start from their initial CHARACTER!

(We exploit lexicographic sorting of the suffixes and skip character comparisons that have already been carried out in previous iterations).

THREE HELP ARRAYS for "MEMORIZATION"

- LCP[1, n-1]
- LLCP[1, n-1] } Defined for every triple L, M, R that may arise in the inner loop of a BINARY SEARCH.
- RLCP[1, n-1]

The values in these arrays are:

- LLCP[M] = LCP(SUFF_SA[L], SUFF_SA[M])

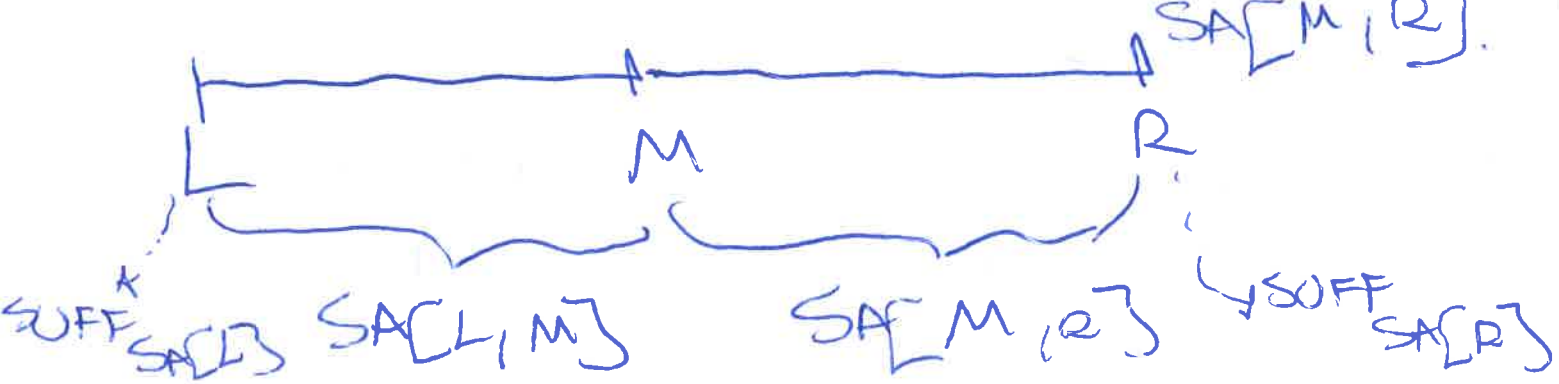
Prefix shared by the left-most suffix SUFF_SA[L] and middle suffix SUFF_SA[M].

- RLCP[M] = LCP(SUFF_SA[M], SUFF_SA[R])

Prefix shared by middle suffix SUFF_SA[M] and right-most suffix SUFF_SA[R]

(RLCP & LLCP can be built in SA[L,R] time via RMQ

=> At every iteration, we must decide whether P lies between SA[L, M] or SA[M, R].



=> To decide where to go next, we need to perform a LEXICOGRAPHIC comparison between ~~SA[L, M] and SA[~~ P and $SUFF_{SA[M]}$

IDEA: Do not start from the first character every time! => would need O(|p|) comparisons!

=> Take advantage of previously computed info.

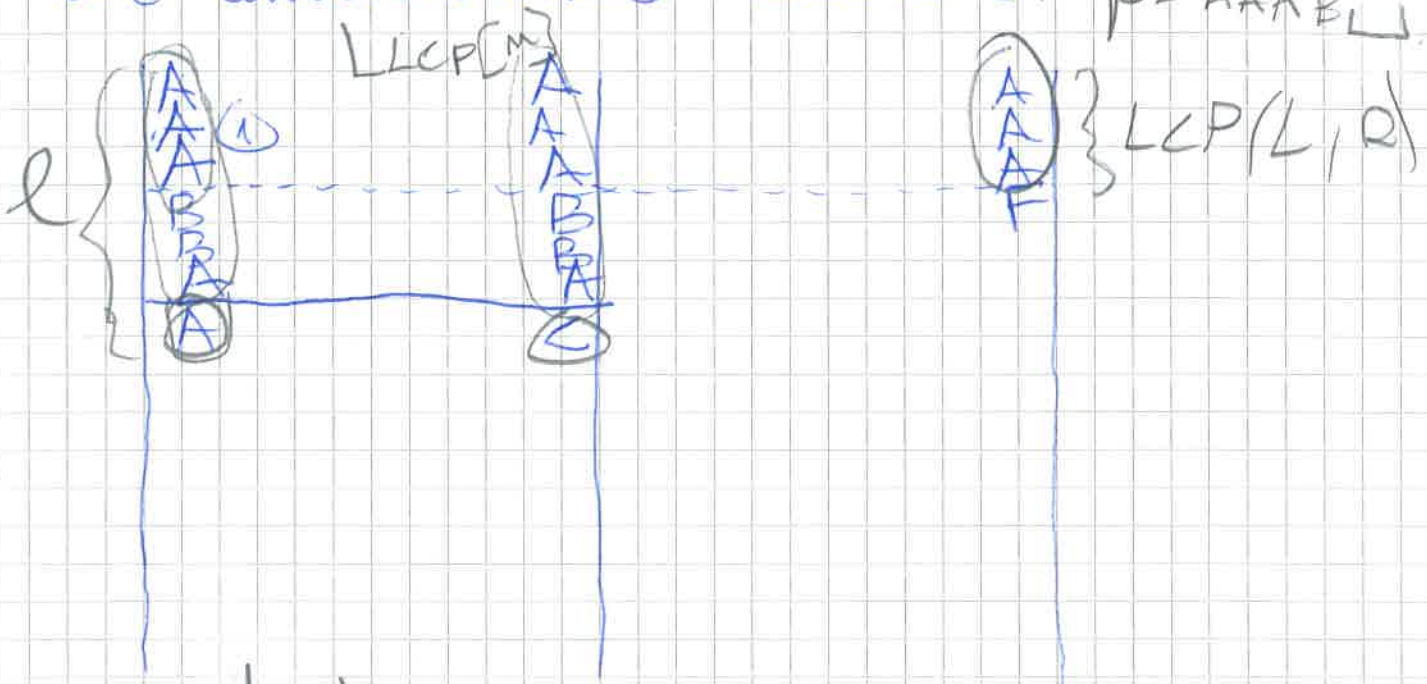
We know that:

P lies between $SUFF_{SA[L]}$ and $SUFF_{SA[R]}$

=> ~~lcp~~ LCP(L, R) is the # characters that are shared by P with its suffixes (as all suffixes in the range must share this # chars).

⇒ We have seen how many characters P shares with L and R!
(LCP[L, R])

We can then have 3 CASES.



$\ell = LCP(P, SUFF_{SA[L]})$ # chars shared with left suffix

$r = LCP(P, SUFF_{SA[R]})$

1) $\ell < LLCP[M]$ ℓ { AAAA, AAAC }

⇒ Then $P > SUFF_{SA[M]}$, and we set:

$m = l$ ⇒ GO RIGHT of M

By INDUCTION $P > SUFF_{SA[L]}$, and their mismatching character lies at $\ell + 1$.

(i.e.: $SUFF_{SA[L]}$ shares more than ℓ characters with $SUFF_{SA[M]}$)

⇒ Mismatch between P and $SUFF_{SA[M]}$
Mismatch between P and $SUFF_{SA[L]}$

⇒ Search continues on SA[M, R] ⇒ BINARY SEARCH on the right of M.

(i.e.: we now need to go right of M)

2) If $l < LCP[M]$ ⇒ BINARY SEARCH on the left of M.

P must be smaller than $SUFF_{SA[M]}$

⇒ M = LLCP[M]

⇒ Search continues on SA[L, M]

(i.e. we now need to go left of M), without extra comparisons.

3) If $l = LLCP[M]$, then P shares l characters with $SUFF_{SA[L]}$ and $SUFF_{SA[M]}$

⇒ Comparison between P and $SUFF_{SA[M]}$ can start from their (l+1)-th character.

⇒ ADVANCE P!

INVARIANT

⇒ Every BINARY-SEARCH step either advances the comparison of P's characters, or it does not compare any characters, but halves the range [L, R] by performing a BINARY SEARCH on a new range.

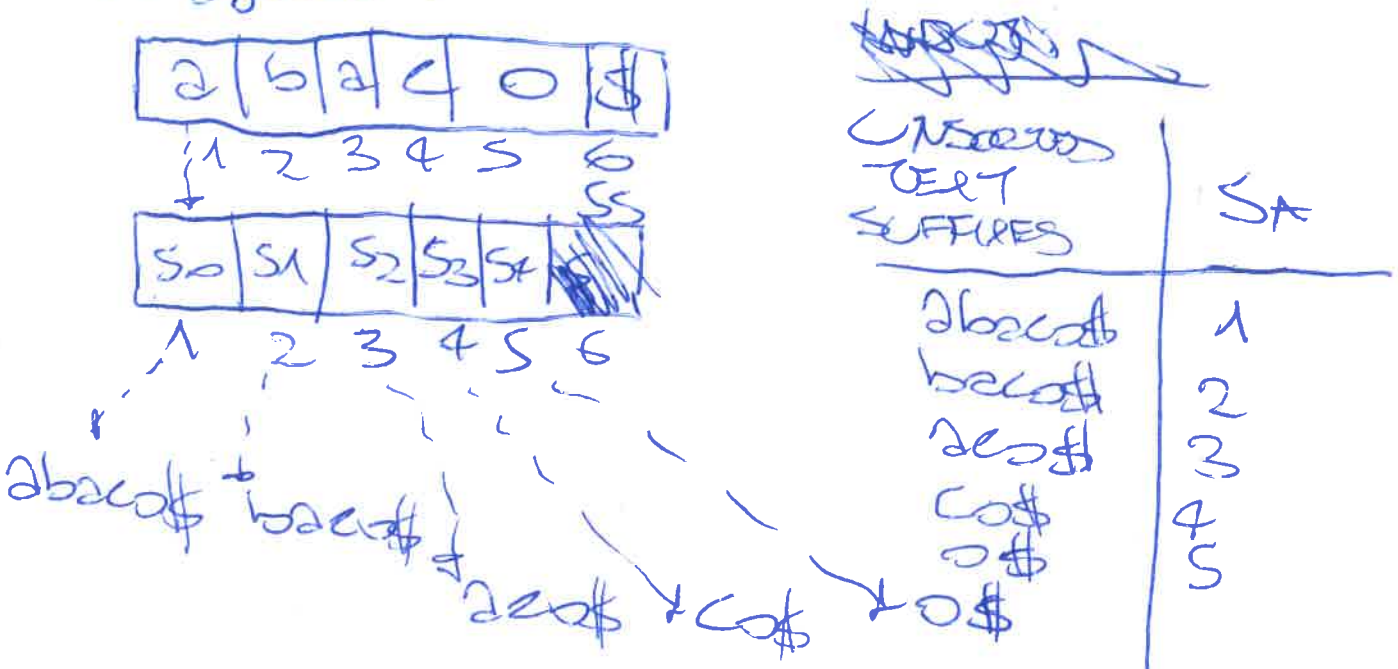
TIME FOR SEARCHING: $O(|P| + \log n)$ (count occurrences of a pattern P in text T)

$O(|P| + \log n + occ)$ (retrieve all the occurrences)

SPACE: $O(n)$

SUFFIX-ARRAY'S CONSTRUCTION:

The suffix array is a sorted sequence of items ⇒ We'll not use a sorting algorithm that is applied over all the suffixes.



⇒ We have made use of a COMPARISON-BASED SORTER (Ex Quicksort) where the comparison function is specialized to compute the lexicographic order between strings (i.e. arrays of characters) ↔ strcmp(SUFFIXES)

↳ We do this in a pretty similar way, comparing character 1 w/ character 2 different suffixes.

#COMPARISONS = $O(n \cdot \log n)$ → each string is n characters long

TIME = $O(n^2 \cdot \log n)$

#I/Os = $O(\frac{n}{B} \cdot n \cdot \log n)$ → n/B I/Os per COMPARISON

SPACE = $O(n \cdot \log n)$ = BITS for SA

```
SA_Construct(char* T, int n, char** SA)
{
  for(i=0; i<n; i++)
  {
    SA[i] = T+i; // new index added
  }
  qsort(SA, n, sizeof(char*), suffix_cmp);
}

SUFFIX_CMP(char** p, char** q)
{
  return strcmp(*p, *q);
}
```

⇒ This procedure is not efficient at all (enough / not. #1/05 & time complexity)

⇒ <u>Steve's algorithm</u> is much more efficient!

CONSTRUCTION OF LCP ARRAY:

* The LCP is defined as:

$$LCP[i] = LCP(SUFF_{SA[i]}, SUFF_{SA[i-1]})$$

BRUTE-FORCE APPROACH:

$SUFF_{SA[i]}$ = abaco

$SUFF_{SA[i]}$ = abacon

⇒ This approach takes $\Theta(n)$ for comparing 2 string with one another.

⇒ For (n) suffixes, this approach takes $\Theta(n^2)$ time.

OPTIMAL LINEAR-TIME ALGORITHM: $O(n)$ (IS THIS REAL?) <TIME, SPACE>

⇒ We need to prove some extra properties & we need to avoid re-scanning the same text characters all the time to come to this result!!

~~By using the fact that the length of~~

Let's refer to a ~~pair~~ consecutive suffixes in the SUFFIX ARRAY: suff$_{i-1}$ and suff$_i$ at positions p and q in SA.

~~T = abcdefbcd~~

SORTED SUFFIXES	SA	SA Positions	LCP
abcdefabchi	j-1	p-1	3
abchi	i-1	p → suff i-1?	
bcdefabchi	j		2
bchi	k	q-1	3
bchi	i	q → suff i	

Let's consider two consecutive suffixes suff i-1 and suff i at positions p and q.

T = abcdefabchi

Assume: we know inductively the value of

LCP[p-1] = LCP(SA[p-1], SA[p])

⇒ Want to show that:

LCP[q-1] = LCP(SA[q-1], SA[q])

can be computed without re-examining the whole SA[q-1], SA[q], but ~~by comparing~~ can start where ~~LCP[p-1] and SA[p]~~ the comparison of SA[p-1] and SA[p] ended.

We need 2 PROPERTIES.

(1) ∀ position x < y, it holds that:

LCP(SUFF SA[y-1], SUFF SA[y]) ≥ LCP(SUFF SA[x], SUFF SA[y])

(i.e.: any ~~suffix~~ preceding suffix to y will certainly share fewer characters than an adjacent suffix to y)

Proof. Prefixes are ordered!

=> MEANING: The farther (backwards) you go for X from y, the fewer characters X shares with y.

Ex.: Consider suff $_{j-1}$ and suff $_{i-1}$.

(abcdef = suff $_{j-1}$, abehi = suff $_{i-1}$)

Take: suff $_j$ = bcdef, suff $_i$ = behi

$LCP(p-1) = LCP(SUFF_{SA[y-1]}, SUFF_{SA[y]})$ (abcdef, abehi)

$\geq LCP(SUFF_{SA[x]}, SUFF_{SA[y]})$ (bcdef, behi)

=> 3 ≥ 2

=> The LCP of the following pair-wise suffixes (y and to y-1) will always be ≥ LCP of the suffixes themselves.

② =>

(2) EXPLAINED.

If SUFF$_{j-1}$ and SUFF$_{i-1}$ share some characters (i.e. LCP[P-1] > 0), then:

suff$_{j-1}$ < suff$_{i-1}$ by the LEXICOGRAPHIC ORDER

=> suff$_{j+1}$ < suff$_i$

And moreover, |LCP[suff$_j$, suff$_i$]| = LCP[P-1]-1

=> First shared character is dropped because by the step ahead.

⇒ The further (backwards) you go, the fewer the shared characters with i.

(Proof: This property follows from the fact that suffixes in SA are ordered lexicographically.)

Follows from (1)

(2) If $LCP[SUFF_{SA[j-1]}, SUFF_{SA[j]}] > 0$:

Then: ⇒

$$LCP[SUFF_{SA[j-1]+1}, SUFF_{SA[j]+1}] \geq LCP[SUFF_{SA[j-1]}, SUFF_{SA[j]}] - 1$$

(PREVIOUS SUFFIXES; NEXT SUFFIXES (expected))

MEANING: (1)

abcdef = $SUFF_{SA[j-1]}$ and $SUFF_{SA[i-1]}$ = abchi

bcdef = $SUFF_{SA[j]}$ ⇒ $SUFF_{SA[i]}$ = bchi

⇒ Now take $LCP[P-1]$ & $SUFF_{SA[j-1]}$

- In the case that $LCP[P-1] > 0$: and $SUFF_{SA[i-1]}$

⇒ Since $SUFF_{j-1} < SUFF_{i-1}$ (because LEX. GRAPH. SORTING)

⇒ $SUFF_j < SUFF_i$ (1) and

$LCP[SUFF_j, SUFF_i] = LCP[P-1] - 1$ (DROP first char)

so we "SKIP ahead" by 1 character
from $j-1$ to j
$(i-1 \quad i)$

$\Rightarrow$ The # SKIPPED CHARACTERS will still be
$LCP[P-1]-1$
and the mismatching char. also remains.

Ex: In the case considered:

LCP(P-1) for SUFF$_{SA[i]}$ and SUFF$_{SA[j]}$ (3)
$\downarrow$
After jumping to SUFF$_{SA[i-1]}$ and SUFF$_{SA[j-1]}$
$LCP[P-1]-1$ between $= (2)$.

$\Rightarrow$ We have proved that the computation of $LCP[Q-1]$ can take advantage of what we have computed at $LCP[P-1]$ (at the previous step) (D.E. D)
$\downarrow$
We AVOID RESCANNING!

TIME COMPLEXITY: $O(n)$
to build LCP

[REQUIREMENTS: — Need the SA.
— Need all things to be stored in INTERNAL MEMORY.

SUFFIX TREE. 10-16

It is a COMPACTED TRIE that stores all suffixes of a string $T[1..n]$ where each suffix is represented by a unique path from the root to one of its leaves.

#LEAVES = n

Leaf's representation (efficient)

<STRING POINTER, START_POS, LEN> = $O(1)$ SPACE.

<STR. POINTER, start_pos, end_pos>

#INTERNAL NODES $\leq \Sigma n$ ⇒ Its label signifies the #shared characters among its children.

#EDGES = TOTAL # NODES $- 1$

$= \#LEAVES + \#INTERNAL\ NODES - 1$

$\leq n + \Sigma n - 1$

Each internal node has at least two outgoing edges!

#EDGES $\leq 2n - 1$

Edges are stored alphabetically by their BRANCHING CHARACTERS (non-empty label ~~starting with~~ meaning the prefix shared by its children)

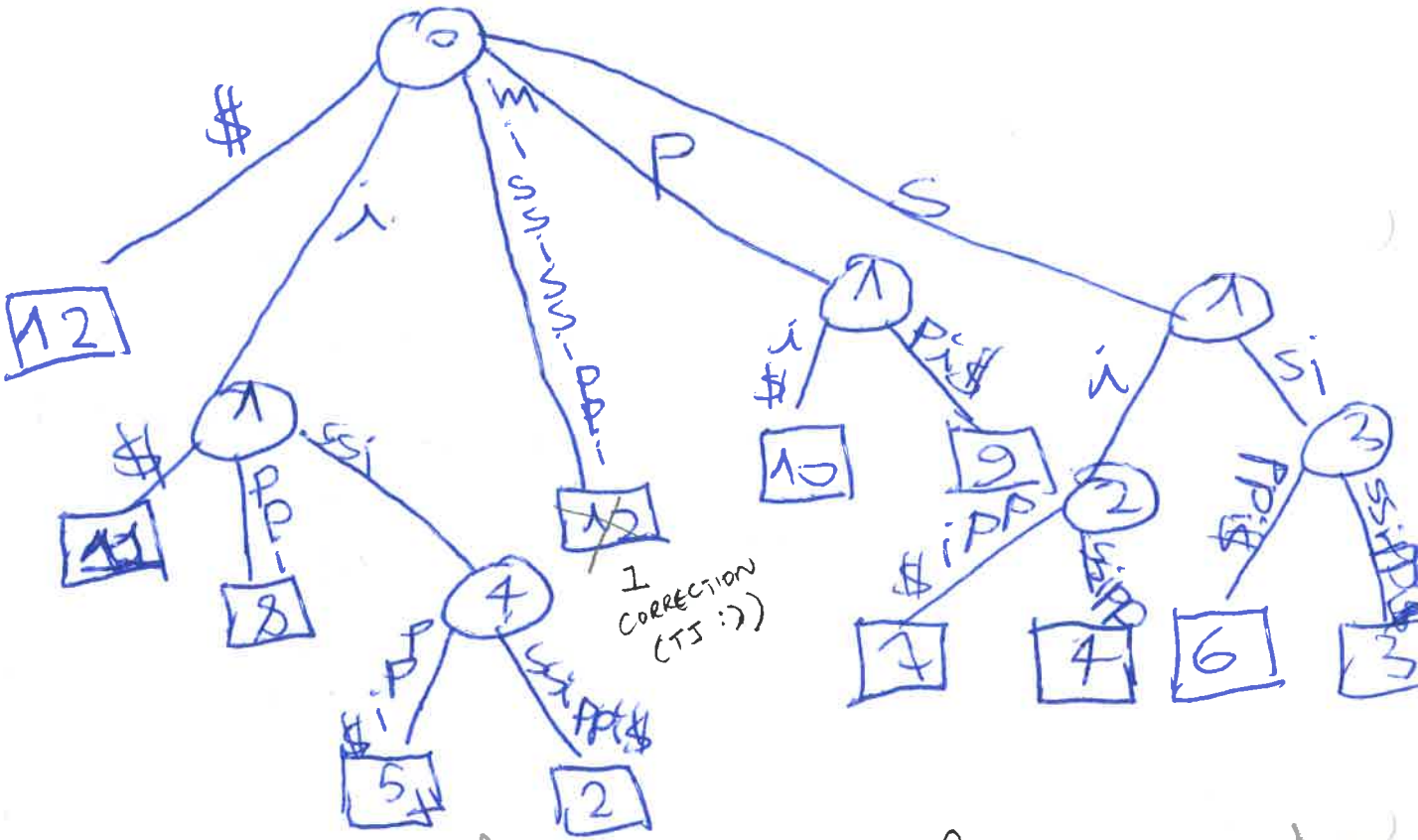
SPACE = $O(n)$ ⇒ TOTAL SPACE (BITS) = $O(n \cdot \log n)$

PURPOSE of a SUFFIX TREE: It is used for efficient storage of suffixes and their efficient retrieval. [QUICK SUBSTRING SEARCH]

EXAMPLE SUFFIX TREE'S CONSTRUCTION.

SUFFIX	SA	LCP	#LINEAR COMPARISONS
$	12	0	1
i$	11	1	2
ippi$	8	1	1
issippi$	5	4	4
ississippi$	2	0	1
mississippi$	1	0	1
pi$	10	1	1
ppi$	9	0	1
sippi$	7	2	1
sissippi$	4	1	1
ssippi$	6	3	1
ssissippi$	3	0	1

=> We get the preceding suffix of every single suffix! by Extended LCP by Char-by-Char Comparison.



=> Via KASAI'S ALGORITHM, we build the SA^{-1} which returns for every suffix its position in SA.
=> Get preceding

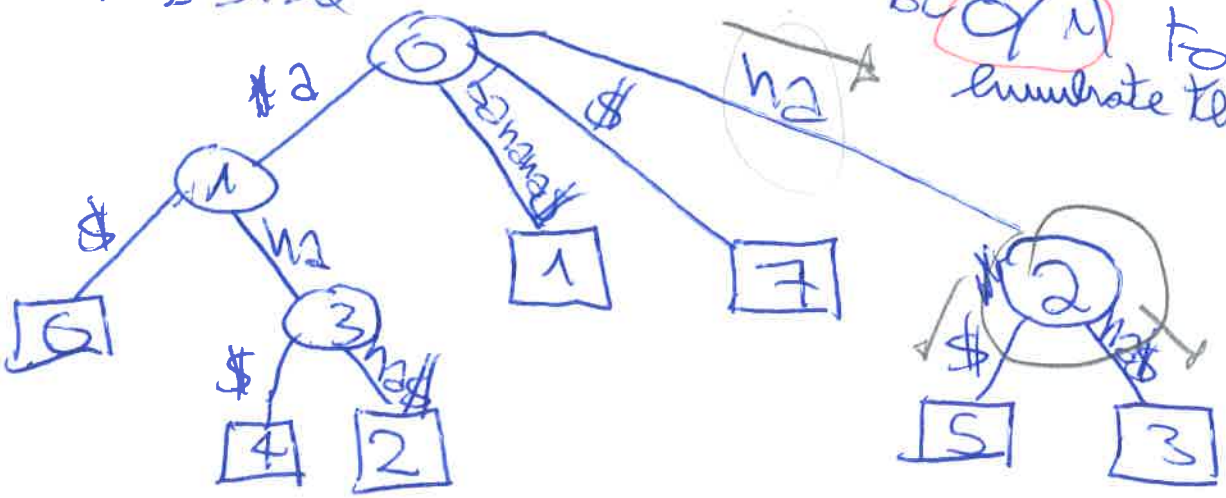
SUBSTRING-SEARCH PROBLEM

We can use the so-built **SUFFIX TREE** for pattern searching $\Rightarrow$ **How?** We match characters along the way of the edges traversed.

(only one downward path is possible!)

If the PATTERN is fully matched, then all nodes descending from the node found identify suffixes prefixed by (P) $\Rightarrow$ $O(\text{occ})$ to list them or $O(1)$ to enumerate them.

$T = b\,a\,n\,a\,n\,a\,\$$
$\quad\;\; 1\;2\;3\;4\;5\;6$



Ex.: $p = na \Rightarrow 2$ occurrences

$O(p + n_{occ})$

PATTERN SEARCHING COST: $O(p t_\Theta + n_{occ})$

Where t_Θ = time to branch out of a node during tree traversal.

t_Θ can have different costs, based on the implementation adopted.

(1) $t_O = O(\log \Theta)$ PLAIN ARRAY &
branching implemented with BINARY SEARCH

Ex:

(2) $t_O = O(1)$ by PERFECT HASH TABLE.

SPACE in (1) (2) is OPTIMAL = $O(n)$

CONSTRUCT SUFFIX ARRAY & LCP FROM THE SUFFIX TREE:

We first define the:

- LCA = Lowest Common Ancestor (of 2 leaves)

The Lowest Common Ancestor is the lowest node in the tree, which is a parent (of multiple generations too) of BOTH LEAVES.

In the case of a SUFFIX TREE, we have that:

$$LCA(node) \equiv LCP(node)$$

Lowest Common Ancestor among 2 leaves ≡ Longest Common Prefix among 2 leaves

- CONSTRUCT SUFFIX ARRAY FROM SUFFIX TREE: LCP AND

The suffix array (SA and LCP) of text T can be constructed based on the SUFFIX-TREE by performing an IN-ORDER-VISIT of the SUFFIX-TREE.

SA:

Whenever a LEAF is encountered, its SUFFIX-INDEX is WRITTEN to the SUFFIX ARRAY, along with the SUFFIX ITSELF.

TIME for SA construction = $O(n)$

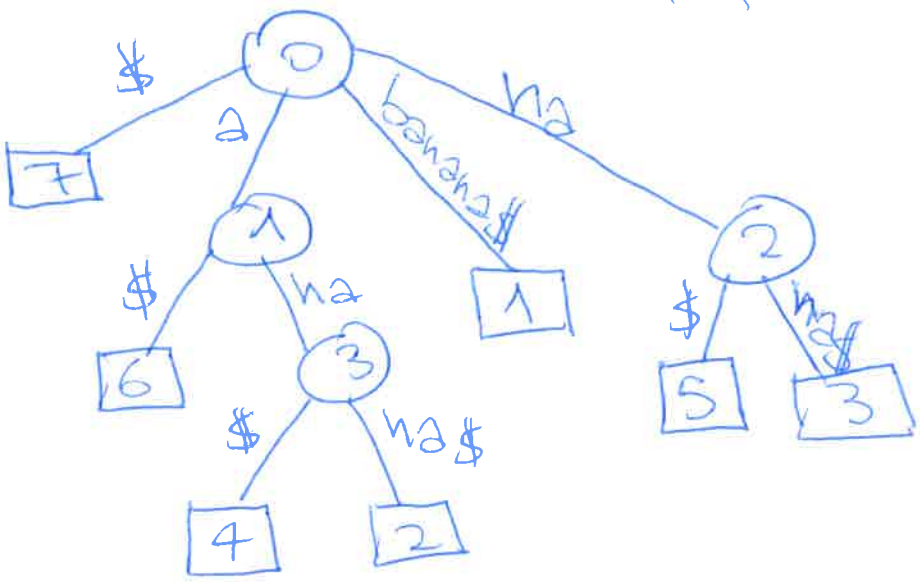
SPACE = $O(n)$

SA + ST

~~EXAMPLE~~

LCP: During the IN-ORDER VISIT, we can write the value of an encountered NODE (leaves) in the LCP of the next visited node.

EXAMPLE: T = banana$
123456 7



=> Perform an IN-ORDER VISIT:

	SUF(T)	SA	LCP	#CHILDREN LIKED	PREFIX
	$	7	0	1	ϕ
	a$	6	1	2	a
	ana$	4	3	3	ana
	anana$	2	0	1	ϕ
	banana$	1	0	1	ϕ
	na$	5	2	1	na
	nana$	3	0	1	ϕ

(SA LEAVES)

ALTERNATIVE APPROACH for LCP CONSTRUCTION.
If we have a DATA STRUCTURE that allows us to perform LCA in O(1) TIME, then we can compute the LCA of all pairs of visit nodes, as:
LCA = LCP

CONSTRUCT SUFFIX TREE from SUFFIX ARRAY (& LCP)

INPUT: SUFFIX ARRAY

OUTPUT: SUFFIX TREE.

"DUMB" APPROACH: Take every single entry of the SUFFIX ARRAY & insert them into the SUFFIX TREE $\Rightarrow O(n^2)$ time!

SMARTER APPROACH: Make use of the LCP computed so far, avoiding the need to compute it every single time for each pair.

We notice the following:

$$LCP[i] = LCP(SA[i], SA[i-1])$$

i.e.: the LCP of leaf(i) is the LCA of SUFF SA[i] and SUFF SA[i-1].

$$LCA[i] = LCA(\text{SUFF } SA[i], \text{SUFF } SA[i-1])$$

IDEA: Compute the LCP/LCA[i] between consecutive entries of the SA (say SA[i], SA[i-1])

IF $\Rightarrow$ LCP[i] does exist, then we insert the SA[i] and SA[i-1] underneath it. (eventually percolating existing nodes)

IF $\Rightarrow$ LCP[i] does NOT exist, then we just LCP[i] at the RIGHT PLACE (percolating existing nodes) & SA[i] & SA[i-1] underneath it.

Ex $T = banana\$$

SORTED SUF(T)	SA	LCP
$	7	0
a$	6	1
ana$	4	3
anana$	2	3
banana$	1	0
na$	5	2
nana$	3	0

$\Rightarrow$ Construct SUFFIX TREE based on SA.

(1) LCP = 0

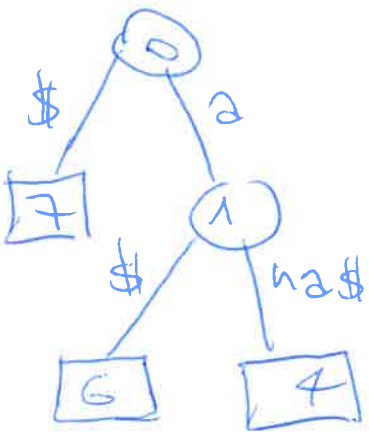
a$ $\Rightarrow$ SA[i] = 6

$ $\Rightarrow$ SA[i-1] = 7

(2) LCP = 1

SA[i] = 4 $\Rightarrow$ ana$

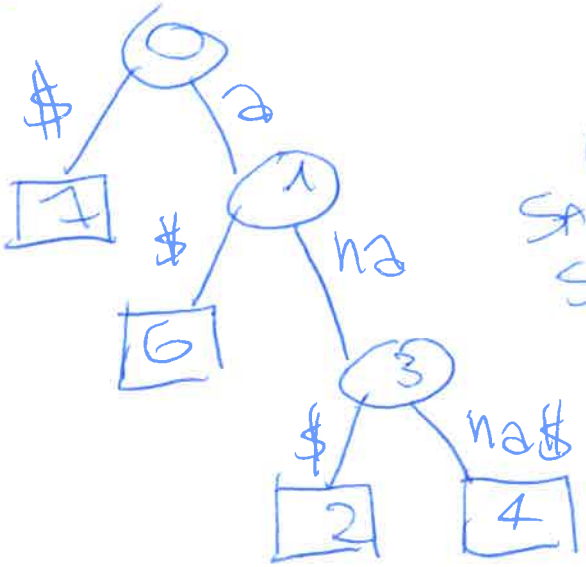
SA[i-1] = 6 $\Rightarrow$ a$



(3) LCP = 3

SA[i] = anana$ = 4

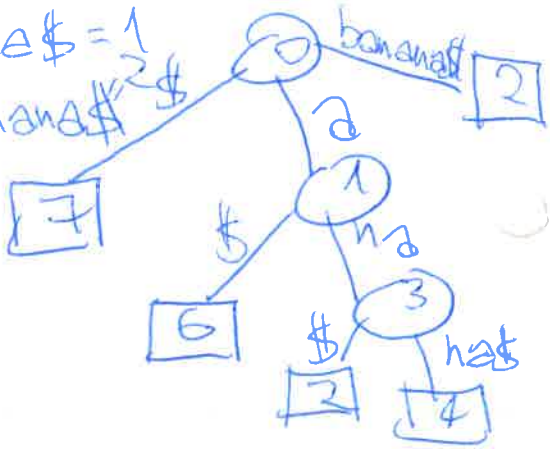
SA[i-1] = ana$ = 2



(4) LCP = 0

SA[i] = banana$ = 1

SA[i-1] = anana$ = 2



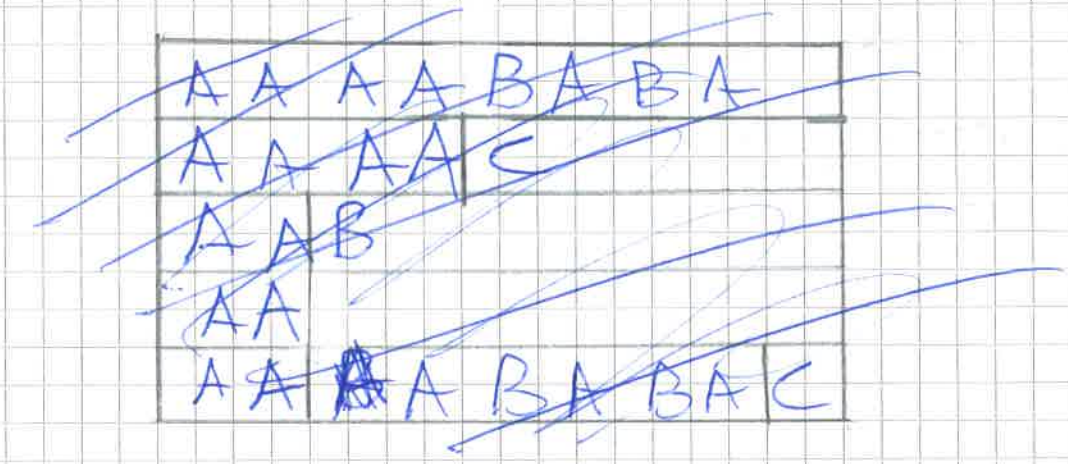
SEARCHING IN A SUFFIX TREE - PROPERTIES:

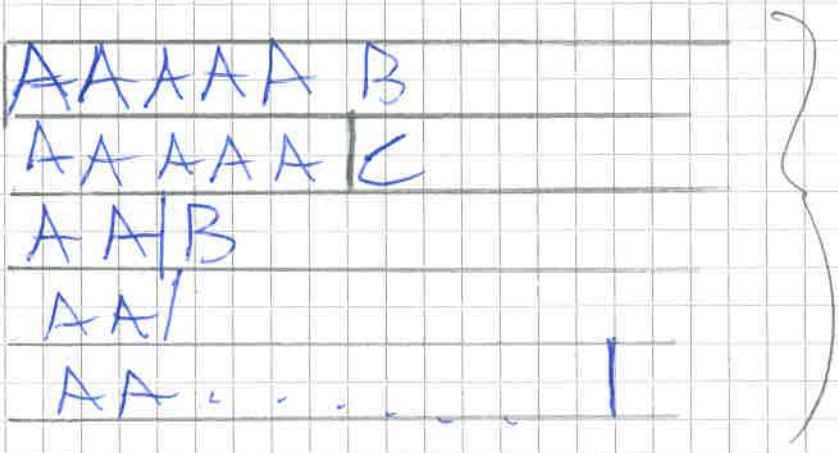
(1) LCP between SA[i] and SA[j].

=

min(LCP[i, j-1])

WHY? Let's visualize the situation:





NB: No property voted actually!

ALPHABETICAL ORDER

⇒ Because of "TRANSITIVITY" & SORTING, we do have this property to hold.

(2) Ex:

i	LCP	SA	SUF(i)
0		12	$
1		11	i$
2	1	8	ippi$
3	4	5	issippi$

(2) ∃ a SUBSTRING (of any length) that repeats ≥ 2 times? (|= ∧)

(do we have a node from that we have ≥ 2 descending LEAVES?)
⇒ LINEAR VISIT To find

(3) ∃ a SUBSTRING of length ≥ 1 that Repeats ≥ (at least 2×)

Ex: ~~WWW~~ Want a SUBSTRING of length at least 4.
MATCH

⇓

i.e.: do we have an internal node labelled at least with (4)?

⇒ LCA = LCP ≥ 4 ?

⇒ Internal node!

X — α — | } Strings share at least α characters!
— α — |

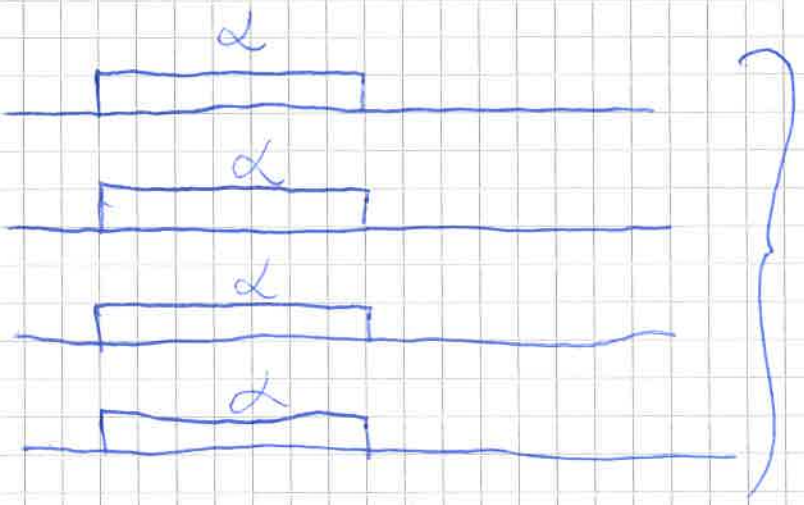
⋮

y — α — |

⇒ Take all strings where LCP ≥ k

(4) ∃ a SUBSTRING of length ≥ 1 that repeats ≥ c times?

⇒ We have (c) strings starting with α.



⇒ Search for a (BLOCK) of c-1 strings, where:

LCP $\geq L$.

(at least c-1!)

Ex. APPLICATION: Intrusion detection ~~system~~ based on SYSTEM LOGS.

Ex: leng ≥ 3
occurs ≥ 2

LCP	SA	SUF
0	12	$
1	11	i$
1	8	ippi$
4	5	issippi$
0	2	ississippi$

CLASS EXERCISE

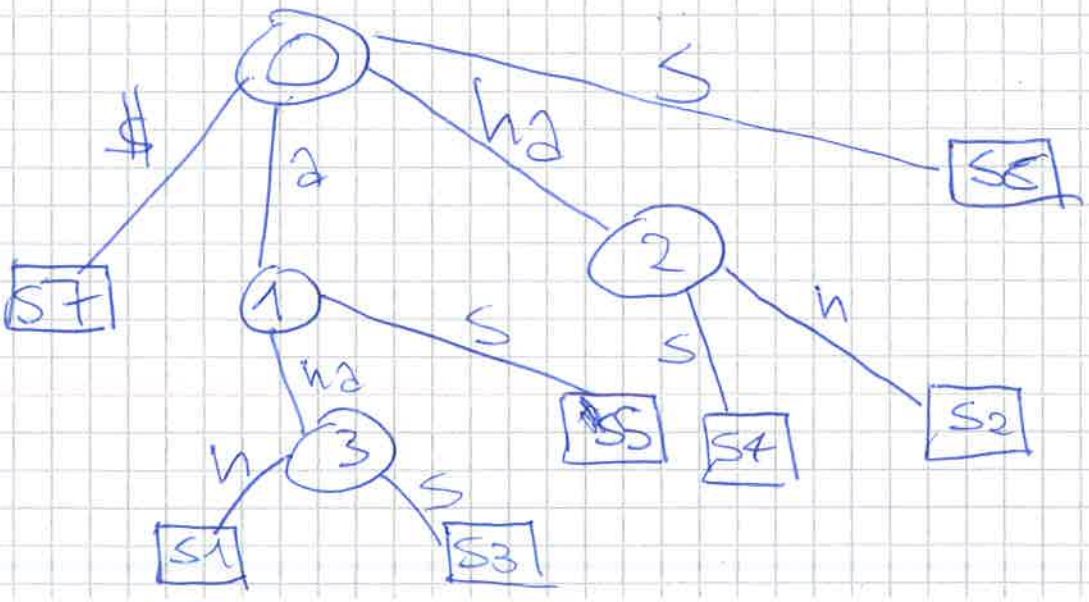
T = ananas$
1 2 3 4 5 6 7

(1) Build a SUFFIX ARRAY.

	SA	SUF(T)	LCP	#COMPS. (LINEAR)	#COMPS. (BRUTE-FORCE)	LCP (Naive)
✓	7	$	0	1	1	0
i-1	1	ananas$	3	~~7~~	4	2
j-1	3	anas$	1	1	2	1
	5	as$	0	1	1	0
	2	nanas$	2	1	3	1
j-1	4	nas$	0	1	1	1
	6	s$	0	1	1	0
		~~TOTAL~~ #COMPS:		10	17	

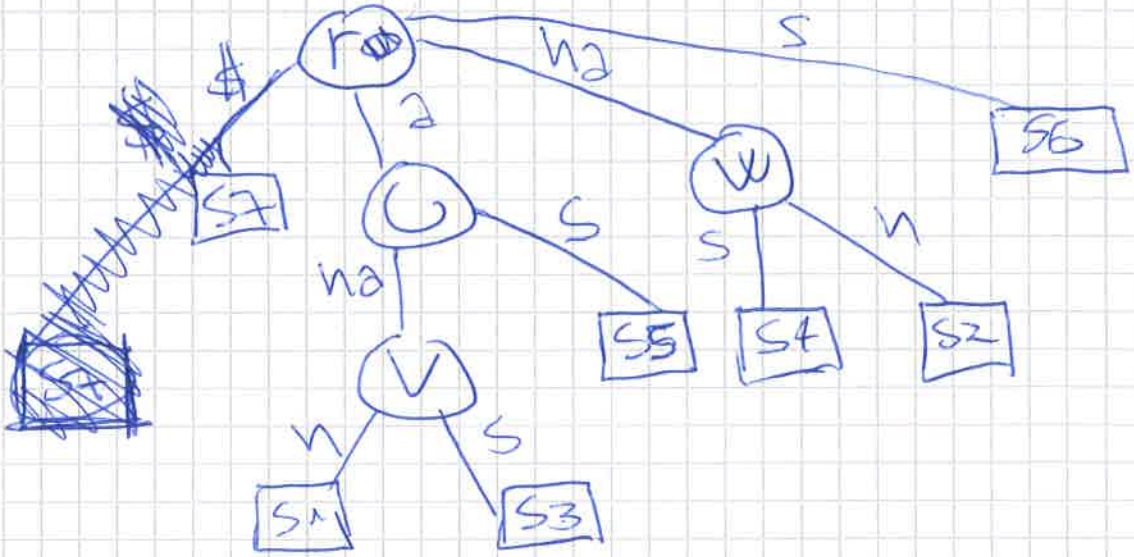
Store previously - found shared PREFIXES (here, the more repetitive things are, the more comparisons we save)

(2) Build a SUFFIX TREE.



⇒ We now want to be able to solve RMQs - Range Minimum Queries over the SUFFIX TREE.

1 Build an EULER TOUR



2 ~~Build~~ Make an EULER TOUR over the TREE:

E_T:	r	$	r	U	V	1	V	3	V	U	S	U	r	W	4	W	2	W	r	6						
D_T:	0	1	0	1	2	3	2	3	2	1	2	1	0	1	2	1	2	1	0	1						
SIGN:	0	+	−	+	+	+	−	+	+	−	+	−	−	+	+	−	+	−	−	−						
MIN:							3		2		1															

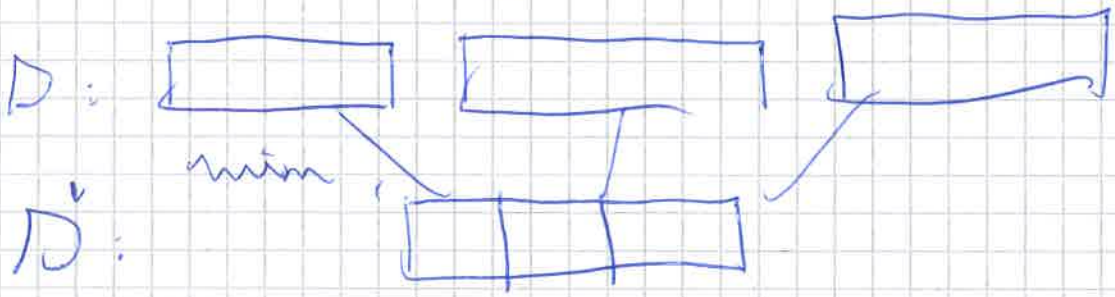
BUCKET SIZE $= \log_2 \frac{n}{B} = 3$

⇒ Goal: find LCA(1,6) ⇒ solve RMQ over the depth.

3 ⇒ Build a sparse table, which contains the MIN. of ranges that are powers of 2.

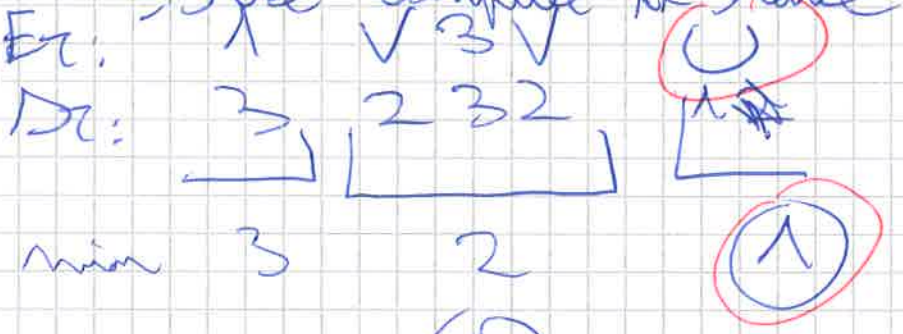
$$\min_{i/k} \{ D'[i : i + 2^k - i] \}$$

NO NEED for a query inside a block.



=> We compute it here in this way.

Ex:



min 3 2

=> 2.

LCA(1, 5) = 2!

APPROXIMATE PATTERN MATCHING

PROBLEM STATEMENT:

"Find all substrings of a text $T[1,n]$ that match a pattern $P[1,p]$ with at most k errors"

Ex: $T = cababa$

$P = baba$

If $k=0 \Rightarrow$ NO occurrence of P found in T.

$k=1 \Rightarrow$ 1 occurrence of P found in T.

K-MISMATCH (i.e.: 1 error occurring)

$\Rightarrow$ Errors denote SUBSTITUTIONS

NAIVE SOLUTION (BRUTE-FORCE)

Try to match pattern P with every possible substring of T, taking the substrings that have at most k-mismatches with the pattern.

Ex: $T = cababa$ #MISMATCHES $P = baba$

All substrings of length P:

- $s_1 = caba$ — 2
- $s_2 = abab$ — 4
- $s_3 = baba$ — 1

MY CANDIDATE

$\Rightarrow$ This approach takes $O(n \cdot p)$ TIME.

(n = # SUBSTRINGS)

NB: Rather inefficient approach, as we need to fully compute the PATTERN every time.

$\Rightarrow$ NEW APPROACH.

INEFFICIENT!

Do not use previously computed INFO.

RANGE - MINIMUM QUERY (RMQ)

⇒ Approach during the K-mismatch search in $O(n \cdot k)$ time [Provided we can solve LCA Queries for P and a substring of T in $O(N)$ TIME]

STEPS INVOLVED:

(1). Build a SUFFIX TREE in $O(n+p)$ TIME from T#P$ DATA STRUCTURE

(2) Build LCA (Lowest Common Ancestor) in $O(n+p)$ TIME, allowing to find the LCA (x, y) in $O(1)$ TIME.

(3) Try to align pattern P with a SUBSTRING of $T[1, n]$, so that i or $j-1$ substrings coincide and j characters mismatch.

Ex: C C G T A C G A T C A G
C C G A A C T

MISMATCHES MATCHES interleave !!

⇒ We would like to:

- COMPARE PATTERN P and text T requiring in $O(1)$ time ⇒ To get $O(n \cdot k)$
  - & need to be able to compute LCP in $O(1)$ time.

⇒ Let's analyze the STEPS:

⇒ In order to compare pattern P[1, P] and a substring of T [after being aligned as by step 3]

⇒ We would like to perform pattern & TEXT equality in O(1) TIME, toget ~~O(n.k)~~ TIME.

K = #MISMATCHES
n = #SUBSTRINGS

OBSERVATION:

~~P occurs in T with K~~ mismatches

If $T[i, i+\ell] = P[j, j+\ell]$ with K errors is one of the MATCHING SUBSTRINGS, then:

- ℓ is the LCP between $T[i \dots]$ and $P[j \dots]$.

EXAMPLE:

~~LCP~~ LCP(T[1,14], P[1,7]) = CCG

⇒ THE MATCHING SUBSTRING depicting the OBS. is:

$T[\hat{4}, \hat{4}+\hat{3}] = P[\hat{4}, \hat{4}+3] \Rightarrow \ell = 3$

- LCP(T[5,14], P[5,7]) = AC

$T[\hat{6}, \hat{6}+\hat{1}] = P[6, 6+1] \Rightarrow \ell = 1$

Q: How to compute:

LCP(T[i+j-1, n], P[j, P]) in O(1) TIME

⇒ LCP(T[7,14], P[4,7])

⇒ We know that SUFFIX TREES and SUFFIX ARRAYS contained some INFORMATION about the LCP!

↳ We are interested in SUFFIXES & P and T through!

↳ We construct the SUFFIX ARRAY or SUFFIX TREE over the string:

$$X = T \# P \$$$

where # is a character not occurring anywhere else

↳ Through this newly constructed data structure, we are hence able to perform LCP computation among SUFFIXES

⇒

(LCA) LOWEST-COMMON ANCESTOR & LCP:

From this Figure, we notice straightaway that there is a STRONG RELATION between the LCP problem over X's suffixes and the LCA in the SUFFIX TREE → the same can be found in the SUFFIX ARRAY.

Ex: Want to find

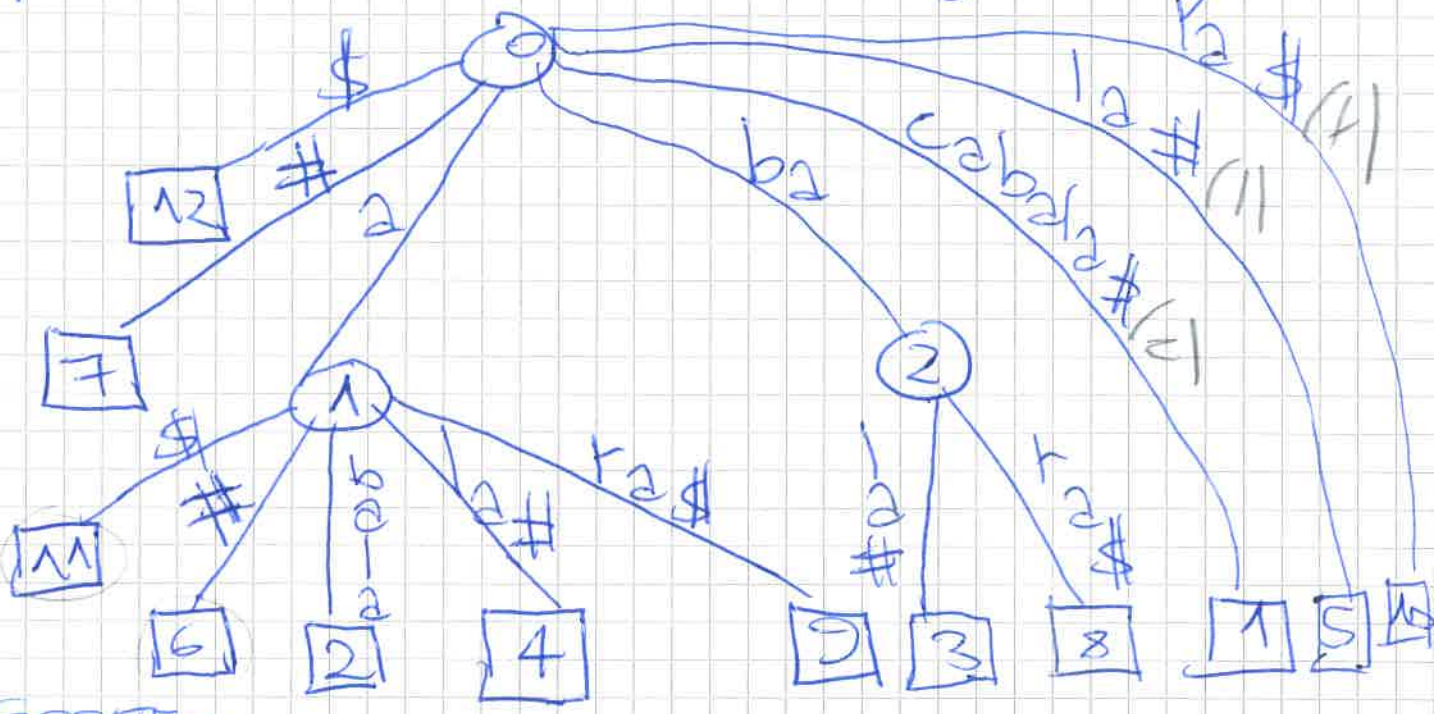
$$LCP(X[i,|X|], X[j,|X|]) = LCA(X[i,|X|], X[j,|X|])$$

T = cabala#bara$

1 2 3 4 5 6 7 8 9 10 11 12

NB: $ < #

1) BUILD A SUFFIX TREE from T:



SORTED SUF(T)	SA	LCP
$	12	0
#bara	7	0
a$	11	1
a#bara	6	1
abala#bara$	2	1
ala#bara$	4	1
ara$	9	0
bala#bara$	3	2
bara$	8	0
cabala#bara#	1	0
la#bara$	5	0
ra$	10	0

(3) Provided we have a DATA STRUCTURE, through that we may find the LCA in O(1) time => we can:

Find the LCA among 2 SUBSTRINGS (& also the LCP's length!)

~~=> By doing this, we would be able to say that the character found will differ at position i~~

EXAMPLE APPROACH:

T = CA BALA #

P = BARA $

=> Look for the LCA of BARA$ and BALA# (in this MAGICAL data structure)

BALA# → [3]

BARA$ → [8]

⇓

DS returns (2) = α

BALA#

BARA$

α

⇓

Surely the character after the LCP will differ!!

⇓ Find [L] and [R]

Compare A# and A$ by looking for their LCA

⇒ [We have a DATA STRUCTURE with that we can find the LCA in $O(1)$ time].

① APPROACH: Solve in $O(n \cdot k)$ TIME via LCA-Computation over the SUFFIX TREE.

↳ Ex. T = cabata#; $K = 1$

P = bata$

⇒ We look for the LCA of bata$ and bala# in the MAGICAL LCA DATA STRUCTURE.

(1) $LCA(|bala\#, |bata\$) = 2 = \alpha$.

⇒ We know that the character following α will surely differ!

bala# , bata$

surely differing

We are now left with performing one LCA query over a# and a$

(2) $LCA(a\#, a\$) = 1$

⇒ We have found that T and P have share 3 characters, and the $K=1$ MISMATCH is satisfied.

↳ #QUERIES performed $= K + 1$

#A TIME COMPLEXITY = $O(n.k)$

#QUERIES over LCA = $O(k)$

⇒ Thanks to these 2 comparisons in hence (1).

⇒ We were looking for (1) MISMATCH & not did 2 QUERIES

⇓

In general, we make $k+1$ QUERIES over n characters at most.

⇒ $O(n \cdot k)$ = TIME COMPLEXITY

#QUERIES over LCA = $O(k)$

(2) HOW TO BUILD the LCA DATA STRUCTURE, TO find LCA in $O(1)$?

[Among 2 different substrings].

RMQ: Range - Minimum Query PROBLEM.

Given an array $A[1,n]$ of elements drawn from an ordered universe, build a data structure RMQ that is able to compute efficiently the POSITION of the minimum in $A[i,j]$

$\forall i,j \in [1,n]$

POSITION of minimum in $A[i,j]$ $\forall i,j$.

Ex: A

1	2	3	4	5	6	7	8	9
7	2	9	3	10	8	20	5	1

i j

RMQ in $A[i,j] = 4$

BRUTE-FORCE (TABLE-BASED) APPROACH:

(To get $O(1)$ TIME) in the SEARCH PHASE

⇒ Store the index of a MIN. entry for each possible combination of i, j.
(just pre-compute all answers)

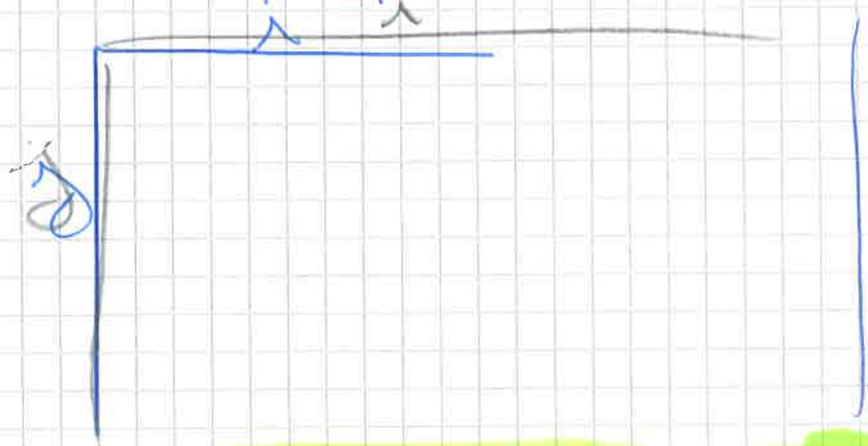


TABLE-build time $= O(n^2)$

TABLE-SPACE needed $= O(n^2)$

QUERY TIME $= O(1)$

SPARSIFICATION APPROACH. $O(n \log n)$

Exploit ASSOCIATIVITY of the MIN. operation ⇒ We can hence try to decompose terms OF [illegible] & apply the MIN onto different pieces, then put 'em all together.

OBSERVATION: Any range (i, j) can be decomposed into two (possibly overlapping) ranges whose size is a power of 2.

⇒ DIMENSION 2e $\log n$ POSITIONS at each (n) position

$$\min\left((i, i+2^L-1), (j-2^L+1, j)\right)$$

Where $L = \lfloor \log(j-i+1) \rfloor$

IDEA: We "SPARSIFY" the quadratic-sized table by storing only ranges whose size is a power of 2.

$\forall$ position (i), we have store an answer to the query:

$RMQ(i, i+2^L)$ by storing $\min(i, i+2^L)$ in the SPARSIFICATION TABLE.

SPARSIFY IDEA: At every element's position in the SPARSIFICATION TABLE, we pre-compute the result of the MIN over its logarithmic-sized ~~range~~ SUBSET ELEMENTS

$\Rightarrow$ We will then have pre-computed all values to answer any RMQ-QUERY.

$$RMQ(i, j) = \min\left[RMQ(i, i+2^{L-1}), RMQ(j-2^L+1, j)\right]$$

Where $L = \lfloor \log_2(j-i+1) \rfloor$

SPACE (for SPARSIFICATION TABLE) $= O(n \cdot \log n)$

QUERY TIME $= O(1)$ (as we have pre-computed all possible results)

SPACE $O(n \cdot \log n) \neq$ OPTIMAL & $O(n)$ SPACE)

OPTIMAL SOLUTION — TWO-FOLD REDUCTION:

In order to get SPACE = $O(n)$ and
QUERY TIME = $O(1)$

⇒ We need to perform the following 2 REDUCTIONS:

a) RMQ COMPUTATION over L2P ARRAY → LCA COMPUTATION over CARTESIAN TREES.

b) LCA COMPUTATION over CARTESIAN TREES → RMQ COMPUTATION over a BINARY ARRAY.

a) RMQ PROBLEM ⇒ LCA OVER CARTESIAN TREES

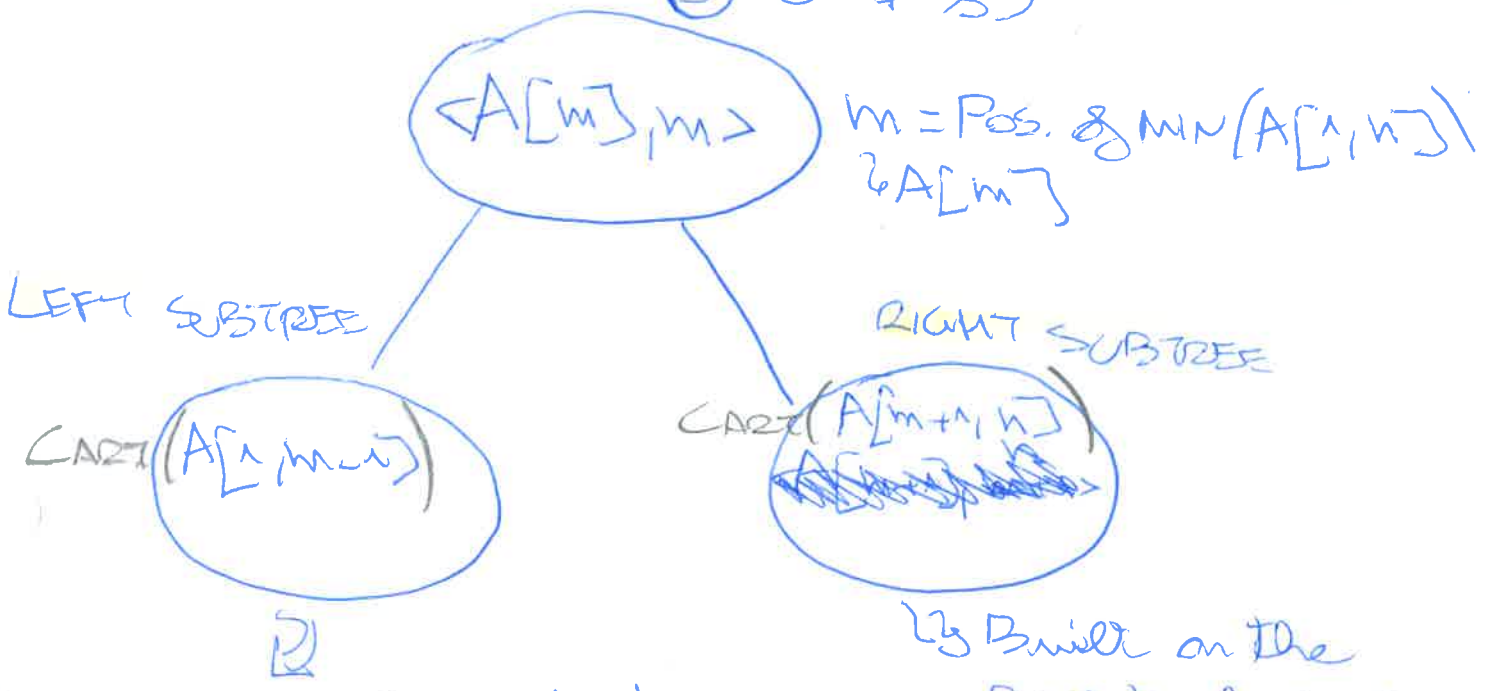
In this step, we construct the CARTESIAN TREE based on the entries of array A[1,n] in a "RECURSIVE FASHION".

↳ CARTESIAN TREE: Binary tree made of (n) nodes, where each node is labeled with:

< VALUE, position >

A = | 7 | 2 | 9 | 3 | 10 | 8 | 20 | 5 | 1 |
1 2 3 4 (5) 6 7 8 9

Different SUFFIXES!

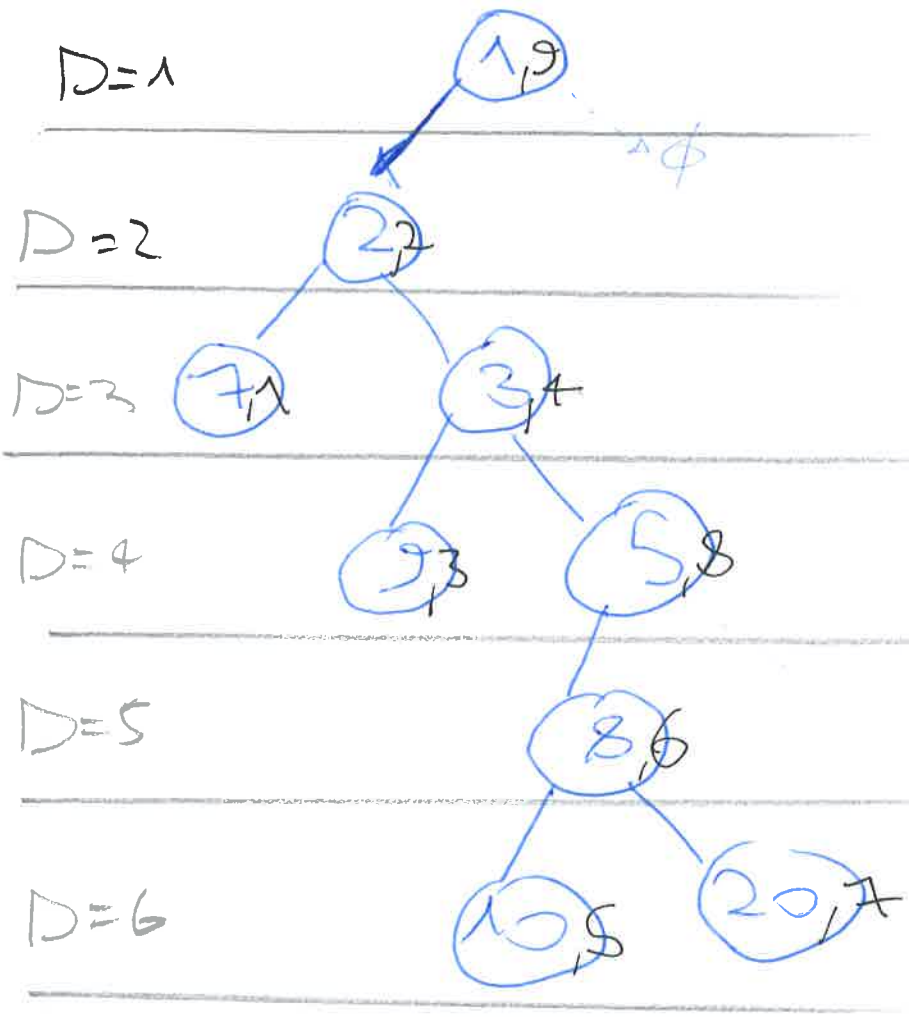


⇒ $RMQ_{LCP}(i, j) = LCA(i, j)$ over the CARTESIAN TREE.

NB: Queried nodes (i) and (j) may be internal nodes as well

EXAMPLE - CARTESIAN TREE'S CONSTRUCTION:

1	2	3	4	5	6	7	8	9
7	2	3	3	10	8	20	5	1



<VALUE, POSITION>

Ex. queries:

RMQ(10,20) = 8,6 = LCA(10,20) (one of the existing minima)

RMQ(7,3) = 2,2 = LCA(7,3)

RMQ(9,5) = <3,4>

VALUE POSITION.

⇒ EULER TOUR'S ARRAY'S LENGTH = [2e - 1]

b) LCA over CARTESIAN TREE (EXTREMA of BLOCKS only) / ADJACENT BLOCKS ⇒ RMQ PROBLEM over a BINARY ARRAY

⇒ We build a BINARY ARRAY, which will allow us to (finally) solve the RMQ problem with $O(n)$ SPACE.

We do this according to the following steps:

1) Compute the EULER TOUR E_T of the CARTESIAN TREE, writing down each node's value when it's visited. (i.e. we perform a PRE-ORDER visit). A node can be visited multiple times too!

2) From the EULER TOUR E_T, build the DEPTH array D_T, which stores the depth of each visited node of the Euler tour.

$$LCA(i,j) = \min_{i'} D[i',j']$$

OVER the CART. TREE

⇒ Node of MIN. depth in the EULER tour.

i' = POS. of first occurrence of i in the EULER tour.

j' = POS. of last occurrence of j in the EULER tour.

⇒ This transformation allows us to re-write:

$$LCA(i,j) = RMQ_D(i',j')$$

SPACE for EULER TOUR + DEPTH ARRAY = $O(n)$

TRANSFORMATION TIME (LCA → RMQ D) = $O(n)$, by storing first & last occurrence

3) Build the "+" and "-" ~~arrays~~

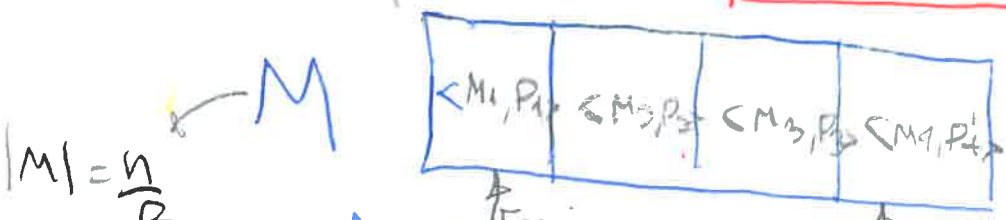
BUILD the MIN. TABLE M

3) We notice that the DEPTH ARRAY D_T has UNITARY VARIATIONS only (hence two consecutive nodes are connected by an edge).

↳ PARTITION D_T into BLOCKS of size b.

where $b = \frac{\log_2 n}{2}$ ⇒ #Blocks = $\frac{|D_T|}{b} = \frac{n}{b}$

For each block, we store the ~~min~~ element in it with min. DEPTH, together with its position.



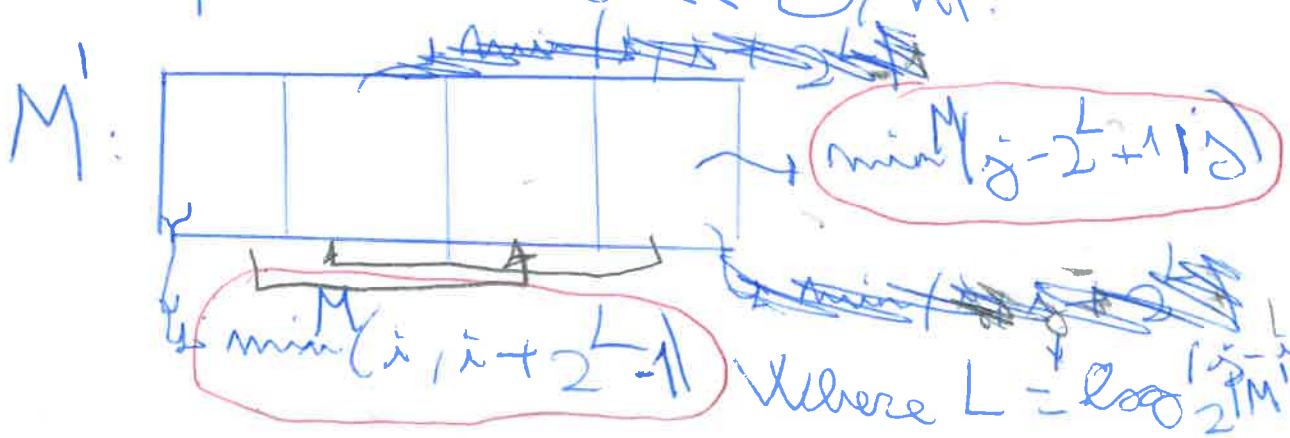
M_K = MIN. of Block K

P_K = Pos. of MIN. in block K

M[K] contains MIN. B_K of D_T.

4) Build on M the SPARSE-TABLE SOLUTION, which solves RMQ queries in O(1) TIME.

NB: M' size is $<< O(n)$.



where $L = \log_2 |M'|$

SPACE FOR M and M' (SPARSE TABLE)

$$O(|M| \cdot \log|M'|) = O\left(\frac{n}{b} \log \frac{n}{b}\right)$$

$$= O\left(\frac{n}{\log n} \log \frac{n}{\log n}\right) = O\left(\frac{n}{\log n}(\log n - \log\log n)\right)$$

$b = \log n \Rightarrow O\left(\frac{n \cdot \log n}{\log n} - \frac{n \cdot \log\log n}{\log n}\right)$

$\Rightarrow O(n)$

Ex. RMQ(i,j) is performed over i,j belonging to different blocks.

→ DOESN'T WORK FOR BSTANCA

1) Ex: 1 2 7 2 | 3 9 3 5 | 8 10 8 20 | 8 5 2 3 2 | 1
   D: 1 2 3 2 | 3 4 3 4 | 5 6 5 6 | 5 4 3 2 | 1
   M: <1,0> | <3,2> | <5,2> | <2,3> | <1,2>

M = [B1: <1,0> | B2: <3,2> | B3: <5,2> | B4: <2,3> | B5: <1,2>]

i=2, j=3

Our query is: LCA(9,5) = 3 (i.e.: 2 query over BLOCKS B2 → B3)

⇒ BUILD the SPECIFICATION TABLE.

B2 | B3

3 | 3

$L = \log(j-i+1)$

$= 1$

$\min M(i : i+2^L-1)$ | $\min M(j-2^L+1 : j)$

$= \min M(2:3)$ | $\min M(4-2+1 : 4) = 2$

$= 3$

⇒ The answer to

$LCA(9,10) = 3$

$\min(B_2, B_3) = 3$

IF i, j ARE IN THE SAME BLOCK:

⇒ Then, the previously described approach wouldn't work.

⇒ We need some extra DATA STRUCTURES to answer RMQs in $O(1)$ time over $i_0, j_0 \in B_i$.

1) Build the EULER TOUR E_T (as before)
2) Build the DEPTH TOUR D_T (as before)
3) Build the MAIN-TABLE M (as before)
4) Build the "+" "−" array S_T, assigning a value based on the value of the previous element.

$$S_T(i) = \begin{cases} \text{"+"} & \text{if } D_T[i] > D_T[i-1] \\ \text{"−"} & \text{if } D_T[i] < D_T[i-1] \end{cases} \quad \forall i \neq 1$$

↳ NB: NOT for the first element in the block!

5) Build the BINARY FINGERPRINTS $F(B_i)$ converting:

"+" ⇒ 1
"−" ⇒ 0

$F(B_i)$ ∀ block (i).

6) RETURN PAIRS:

$\langle F(B_K), \text{pos. min in } F(B) \rangle$

7) Build a LOOKUP TABLE, indexed by all POSSIBLE QUERY OFFSETS i_0, j_0 within the block B_K ⇒ to be able to answer queries in $O(1)$ TIME.

$TL[i_0, j_0, F(B)] = \text{pos. min in } F(B)]$

~~SPACE OCCUPANCY~~

POSSIBLE CONFIGURATIONS & FINGERPRINTS - SPACE REQUIRED:

$\#\text{POSSIBLE CONFIGURATIONS} = 2^{b-1} = O(2^b)$

$= O(2^{\frac{\log n}{2}}) = O(2^{\log (n)^{\frac{1}{2}}}) = n^{\frac{1}{2}} = \sqrt{n}$

$b = \frac{\log n}{2}$ = SIZE of FINGERPRINT

TOTAL SPACE OCCUPANCY (of LOOKUP TABLES)

$O(\sqrt{n} \cdot \log n \cdot \log n) = o(n)$

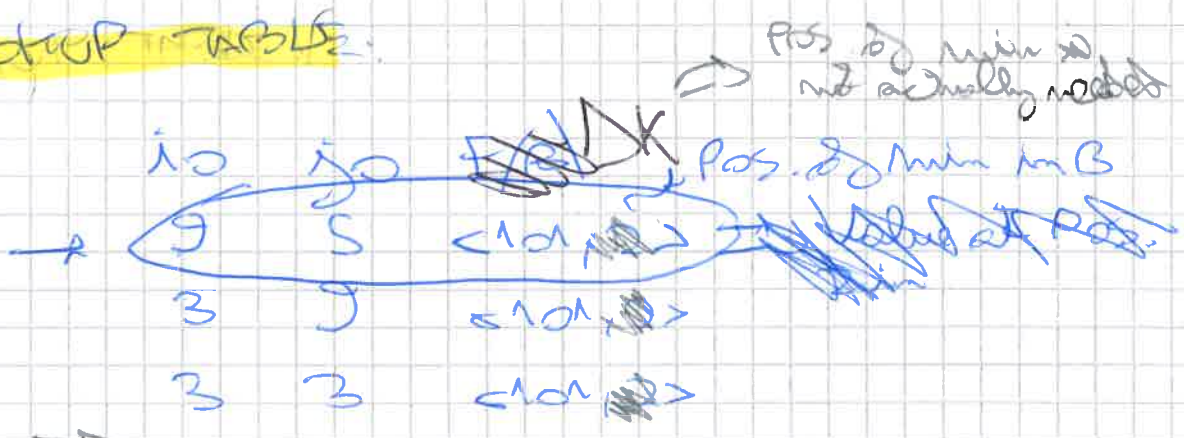
TIME to BUILT the lookup TABLE $= O(n)$

FINAL LOOKUP TIME $= O(1)$

EXAMPLE of an RMQ over i_0, j_0 within the same BLOCK:

	B_1	B_2	B_3	B_4
1) E_T:	1 2 7 2	3 9 3 5	8 10 8 20	8 5 3 2 1
2) D_T:	1 2 3 2	3 4 3 4	5 6 5 6	5 4 3 2 1
3) M:	<1,0>	<3,2>	<5,2>	<2,3>
4) S_T:	1 + + −	3 + − +	5 + − +	5 − − −
5) F(B):	<1 1 0>	<1 0 1>	<1 0 1>	<0 0 0>
6) PAIRS:	<1 1 0, 0>	<1 0 1, 2>	<1 0 1, 2>	<0 0 0, 3>

LOOKUP TABLE:



$\Rightarrow$

$\Rightarrow$ From $<F(B)$, and consequently answer the query $LCA(9, 5) = B_2[2] = 3$.

ANSWER QUERY i_0, j_0 (in same BLOCK)

1) Compute i_0, j_0 wrt. beginning of the Block they belong to.

2) Compute $F(B)$ of that BLOCK.

$\Rightarrow$ Based on $<i_0, j_0, F(B)>$, the minimum can be RETRIEVED.

CLASS
RMQ EXERCISE:

Let $T = abaabaabaa$, $P = abaa$.

a) How does the ≤ 1-mismatch algorithm works on T and P, given that it is provided a DS that gives LCA in $O(1)$ time.

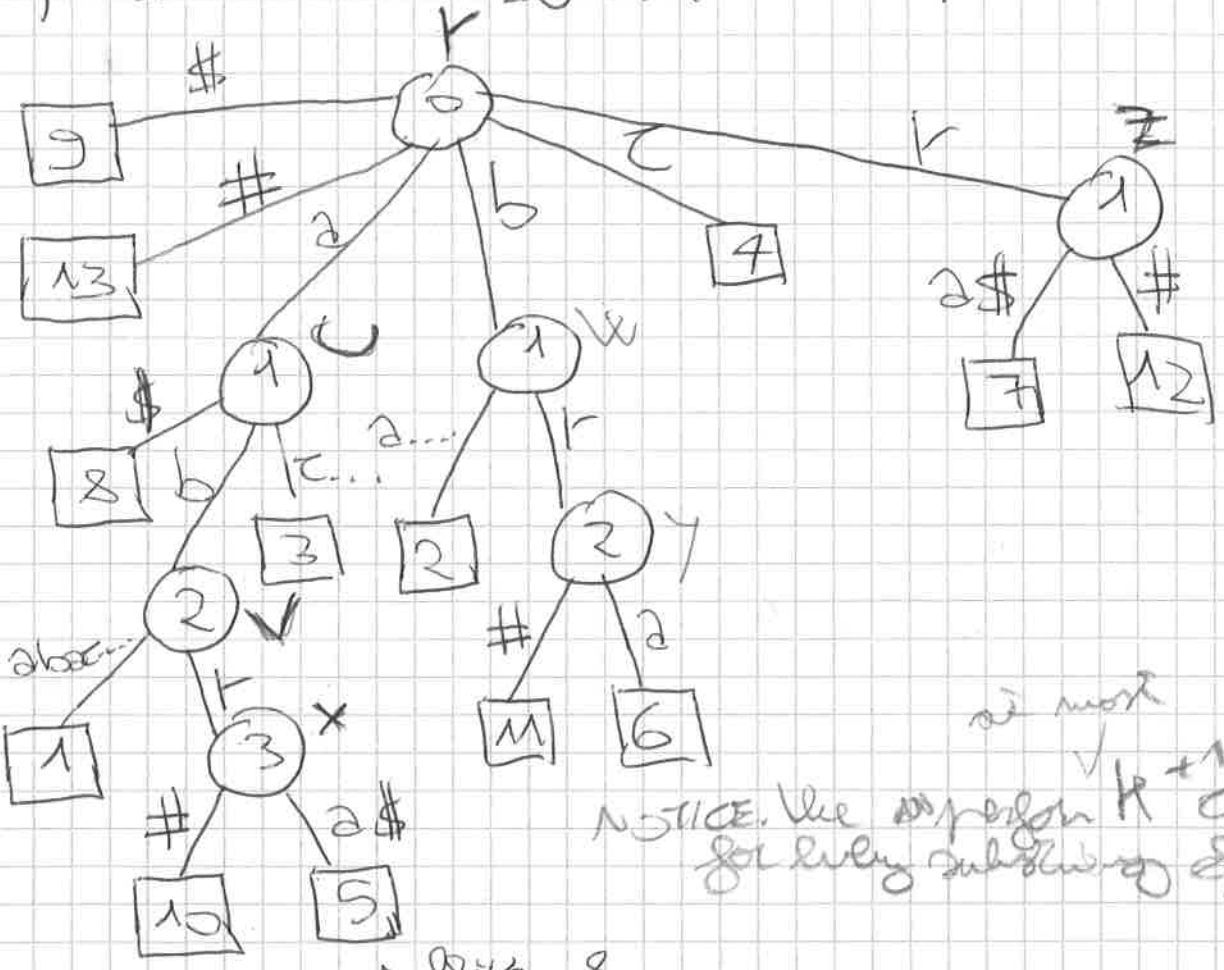
$\Rightarrow$ Need to find 2 1-mismatches (i.e: an LCA ≥ 2 is enough!)

(1) Concatenate T and P

$T\$P\# = abaabaabaa\$abaa\#$

T: 1 2 3 4 5 6 7 8 9 10 11 12 13 — P: 1 2 3 4

2) Build the SUFFIX TREE:



NOTICE: We perform $k+1$ comp. for every suffix of T.

$\Rightarrow$ We compare P with the first $k+1$ suffixes of T.

- P comp. $T[1:\dots]$ $\Rightarrow LCA(1,10) = 2$ ✓ $\Rightarrow$ 1 MISMATCH
  $P: (1,12)$; $T: (2,3)$
- P comp. $T[2:\dots]$ $\Rightarrow LCA(2,10) = 0$ ✗
  ~~ROOT~~ $\Rightarrow LCA(3,12) = 0$ ✗ } STOP, 2 MISM.

P comp. T[3:...] → LCA(3,1) = 1 ⇒ Next char. no a mismatch

↳ Jump to next next char

↳ LCA(5,12) = 0 ⇒ Stop. 2 mismatches!

P comp. T[4:...] → LCA(4,1) = 0

↳ LCA(5,11) = 0 ⇒ Stop. 2 mismatches

P comp. T[5:...] → LCA(5,1) = 3 ✓

~~xxx~~ mismatch in T:(7,8) P:(#,13)

b) Provide a DATA STRUCTURE that allows for LCA in O(1) time

1) Perform EULER TOUR.

~~E_T: #rU8UV1VX1oX5XVU3UrW2Wy11y6y W|-4|-Z7Z12Z|-~~

~~D_T:~~

block size = 4

	B1			B2			B3			B4			B5			B6			B7			B8			B9		
E_T:	r	U	8	U	V	1	V	X	1	o	X	5	X	V	U	3	U	r	W	2	W	y	1	1	y	6	y	W	|	-	4	|	-	Z	7	Z	1	2	Z	|	-
D_T:	0	1	2	1	2	3	2	3	4	3	4	3	2	1	2	1	0	1	2	1	2	3	2	3	2	1	0	1	0	1	2	1	2	1	0					
M:	*	0		2			3			1			0			2			0			0			0					

⇒ We need to solve LCA(10,6)

⇒ Only need the SPARSIFICATION TABLE

Blocks 3, 4, 5, 6

To answer the query given ⇒ LCA(10,6)

⇒ It is hence sufficient to apply SPECIFICATION to the BLOCKS where (10) and (6) lie in.

$L = \log_2(j - i + 1)$

$\log_2(3)$

↳ $\min(i, i + 2^L - 1) = \min(3+0, 3+2^1 - 1) = 1$

↳ $\min(j - 2^L + 1, j) = \min(6 - 2 + 1, 6) = 6$

$k = 0$

⇒ LCA(10,6) = min(1,0) = 0!

[BLOCKS ONLY WITH MULTIPLES OF BLOCKS]

$L = \lfloor \log_2(j - i + 1) \rfloor$

$L =$

SPARSE TABLE - PREFIX / SUFFIX ARRAY

This strategy allows us to answer queries among POSITIONS that are NOT multiples of BLOCKS, i.e. not the starting and ending position of a block, like in the successive approach.

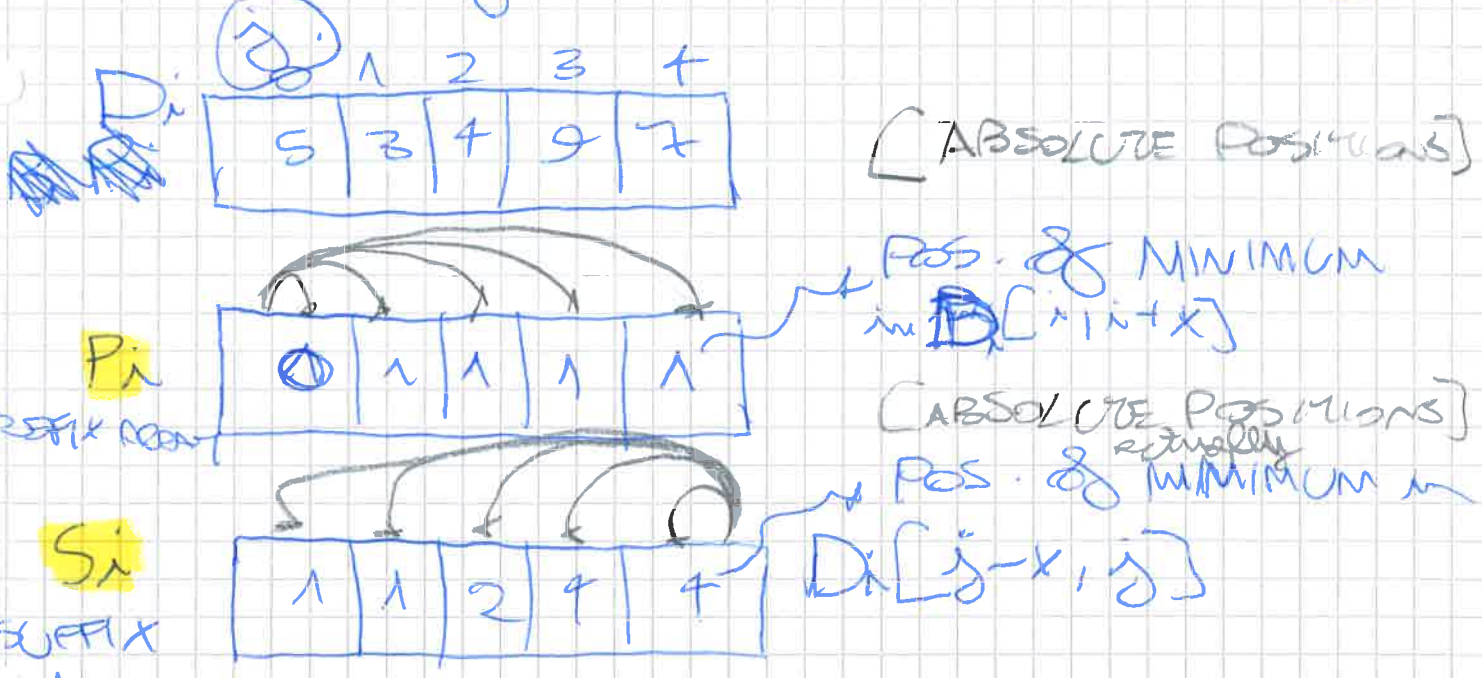
1) Build the EULER TOUR (E-1) over the CARTESIAN TREE, as in (1)

2) Build the DEPTH TOUR (D) as in 2)

3) Build the MIN-TABLE (M)

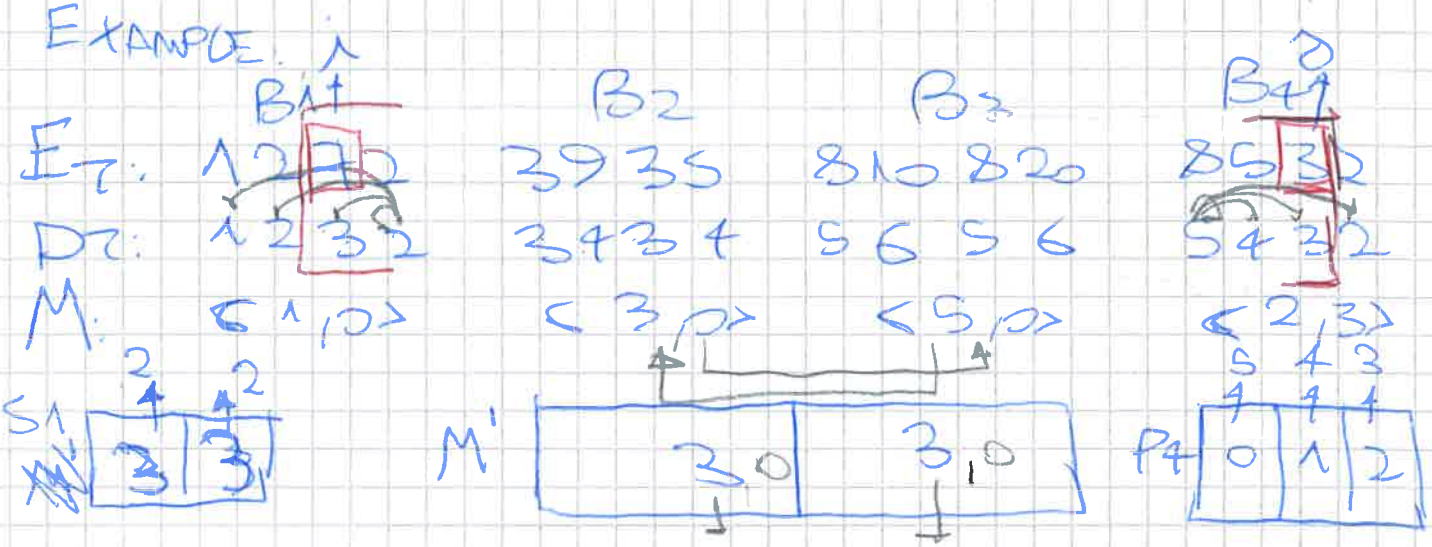
4) Build the SPARSE-TABLE (M') of the ranges in-between blocks i and j

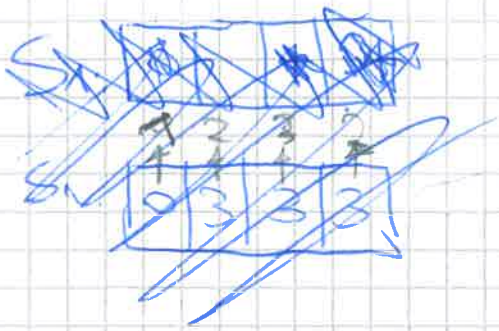
5) Build the PREFIX ARRAY & the SUFFIX ARRAY of the blocks that contain (1) and (2)

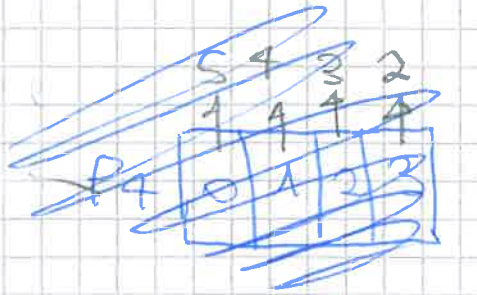


6) Take MINIMUM of the PREFIX ARRAY (left part of SPARSE-TABLE / middle part, OPTIONAL), SUFFIX ARRAY

EXAMPLE.







$$\min(S_1, M', P_4) = 2$$

(11) # INTEGER CODING

PROBLEM: $S = \{s_1, s_2, \ldots, s_n\}$ be a sequence of INTEGERS (not ordered), $s_i \leq N$, possibly repeated

GOAL: Represent integers of S as binary sequences that are SELF-DELIMITING and occupy few bits.

If integers are NOT POSITIVE

↓

We transform - POSITIVE INTEGERS → EVEN POSITIVE INTEGERS

- NEGATIVE INTEGERS → ODD POSITIVE INTEGERS

$T \subseteq \mathbb{Z}$

$x \in T \geq 0 \rightarrow 2x$ (EVEN)

$x \in T < 0 \rightarrow -2x + 1$ (ODD)

NB: Some encoders can encode (0), others cannot.

FIXED-LENGTH REPRESENTATION

(i.e. 4 / 8 bytes to represent an integer)

Take $M = \max_{s_i \in S} s_i$ (i.e. max. integer in the set S)

$\forall x \in S, x \leq M \Rightarrow \#BITS = 1 + \lfloor \log_2 M \rfloor$

SIMPLE BINARY ENCODING! (Ex: 3 → 2 bits)

GOOD WHEN? When integers are more or less of same magnitude

BAD WHEN? Integers are very sparse (lots of bits wasted!)

— BAD.

Not self-DELIMITING code if concatenating multiple integers' encoding [we wouldn't know where an integer starts and the next one ends at].

↳ NO UNAMBIGUOUS DECODING.

Ex: $S = \{1, 2, 3\}$

$C = \{1101\}$ ⇒ How to we decode this? (DUNNO!)

VARIABLE-LENGTH REPRESENTATION

(allow following numbers)

We really like a FAST & SUCCINCT way of coding and decoding, which requires a VARIABLE # BITS for SPACE & is UNAMBIGUOUS to be decoded.

WANT: OPTIMAL ENCODING.

We have an OPTIMAL ENCODING of integer X if

$$\sum_X P(x)\,|ENC(x)| \geq \sum_X P(x) \log_2 \frac{1}{P(x)}$$

($P(x)$ is probability of symbol x occurring)

AVG. # BITS used by an encoder — LOWER-BOUND — ENTROPY (SHANNON'S CODING THEORY)

Whatever the encoding, we can NEVER do better than this! (No Proof!)

Provides an UPPER BOUND to the # BITS used to encode integer X.

SHANNON'S CODING THEORY

The ideal code length $L(x)$ for a symbol x is: $\log_2 \frac{1}{P(x)}$ (ENTROPY) $P(x) = P\{x \text{ symbol occurs}\}$

⇒ UPPER BOUND of H?

$$0 \leq H \leq ?$$

↳ UPPER BOUND!

$$H = \sum_{x \in S} P\{x\} \cdot \log_2 \frac{1}{P\{x\}} \Rightarrow \sum \Rightarrow \text{ENTROPY}$$

where x is a char of S.

$$\sum_{x \in S} \frac{1}{|S|} \cdot \log_2 |S| = \log_2 |S|$$

UPPER BOUND of ENTROPY

$$\Rightarrow 0 \leq H \leq \log_2 |S|$$

ENTROPY

Ex: If you have 16 digits ($|S| = 16$), the entropy will go from 0 to 4.

⇒ ENTROPY grows (locally) with the #DIGITS in S.

H ENTROPY'S MEANING: AVG. # BITS TO encode a string (from a SOURCE) emitting X.

If $H \to 0$: fully compressed, values are concentrated at some values (SURE NOT random)

RANDOM source. ← ($\log_2 |S|$, where values are NOT concentrated)

OPTIMALITY CONDITION / EQUATION

In order for an encoding scheme to be optimal, we must have:

$$|ENC(x)| = \log_2 \frac{1}{P\{x\}}$$

$|ENC(x)|$: AVG. # BITS required for encoding a generic integer x

$P\{x\}$: PROBABILITY that an integer x occurring in the DISTRIBUTION considered

EX:

$S = \{1, 2, 3\}$

$P: \quad 0.5 \quad 0.3 \quad 0.2 \quad = P\{x\}$

BAD ENCODING:

LOTS of BITS!

$1 \rightarrow 000$

$2 \rightarrow 10$

$3 \rightarrow 11$

$|ENC(x)| = 0.5 \times 3 + 0.3 \times 2 + 0.2 \times 2$

$= 1.5 + 0.6 + 0.4$

$= 2.5$ bits

GOOD ENCODING:

$1 \rightarrow 0$

$2 \rightarrow 10$

$3 \rightarrow 11$

$|ENC(x)| = 0.5 \times 1 + 0.3 \times 2 + 0.2 \times 2$

$= 0.5 + 0.6 + 0.4 = 1.5$ bits

⇒ We hence notice that, as we perform a weighted sum based on the encoding length and the P. of an integer occurring

DESIRED PROPERTY {
UNPROBABLE SYMBOL → Long encoding
PROBABLE SYMBOL ⇒ Short encoding
}

OPTIMAL DISTRIBUTION for a FIXED-LENGTH ENCODER:

(i.e. for which distribution is a fixed-length encoder OPTIMAL?)

~~$\forall x,\ \log |Enc(x)| = \log_2 \frac{1}{P\{x\}}$~~

~~$\log_2 \frac{1}{P\{x\}}$~~

Bits needed to encode x

$$|Enc(x)| = \log_2 \frac{1}{P\{x\}}$$

(NOT considering +1)

$$\log_2 M = \log_2 \frac{1}{P\{x\}} \quad \forall x < M$$

{MAX value of size}

$$P\{x\} = \frac{1}{M}$$

⇒ FIXED-LENGTH encoder is OPTIMAL if the DISTRIBUTION is UNIFORM!

UNARY CODE: $(x \neq 0)$

IDEA:
$U(x)$ for an integer $x \geq 1$ is given by a sequence of $x-1$ bits set to 0, ended by a (DELIMITING) bit set to 1.

$$U(x) = \underbrace{000\ldots0}_{x-1}\,\underbrace{1}_{\text{DELIMITING BIT}} \Rightarrow \text{Encode non-zero values}$$

$x \geq 1$; x

SPACE & $U(x) = x$ BITS!
(exponentially more bits than BINARY CODE & FIXED-length representations)
⇒ SPACE inefficient for large x

Ex: $U(5) = \underbrace{0000}_{4\ \text{"0s"}}1$

OPTIMAL DISTRIBUTION & UNARY ENCODING.

$$|ENC(x)| = \log_2 \frac{1}{P[x]}$$

As many bits as x value ← $x = \log_2 \frac{1}{P[x]}$

$$2^x = e^{\log_2 \frac{1}{P[x]}}$$

$$2^x = \frac{1}{P[x]}$$

$$\Rightarrow P[x] = \frac{1}{2^x}$$

18
EXPONENTIAL DISTRIBUTION

ELIAS CODES:

γ and δ ENCODING are UNIVERSAL ENCODES, as their code length is $O(\log x)$ for any integer x, and PREFIX-free.

Also, they are UNAMBIGUOUSLY-DECODABLE, by inverting the CODING PROCEDURE.

γ-(GAMMA) CODE:

Given an integer (x) as input, where $B(x)$ is its BINARY REPRESENTATION and $|B(x)|$ is the # BITS used to represent $B(x)$.

$$L = |B(x)|$$

$$\gamma(x) = \underbrace{000\ldots0}_{L-1\ 0s}\ \underbrace{1 \cdot B(x)}_{L}$$

① ②

$\gamma(x)$ is formed by:

1) The unary representation of $|B(x)|$. (i.e. $|B(x)|-1$ zeros + one 1)

2) $B(x)$, the binary representation of x.

SPACE: $L-1+L = 2L-1 = 2\cdot\log x - 1$

$= O(\log x)$

EXAMPLE

$\gamma(9) \Rightarrow B(9) = 1001$

$\gamma(9) = \underbrace{000}_{|B(A)-1| \text{ Zeros}}\ 1001$

OPTIMAL DISTRIBUTION for γ-ENCODING

$$|ENC(x)| = \log_2 \frac{1}{P\{x\}}$$

$$2 \cdot \log x - 1 = \log_2 \frac{1}{P\{x\}}$$

$$2 \cdot \log_2 x - \log_2 2 = \log_2 \frac{1}{P\{x\}}$$

$$\log_2 x^2 - \log_2 2 = \log_2 \frac{1}{P\{x\}}$$

$$\log_2 \left(\frac{x^2}{2}\right) = \log_2 \frac{1}{P\{x\}}$$

$$P\{x\} = \frac{2}{x^2}$$

DISADVANTAGES:

Need to perform BIT-BY-BIT decoding & encoding (slow!)

☺ Actually work when small values are involved!

δ-ENCODING (DELTA)

It applies γ-code in place of UNARY ENCODING.

$\delta(x)$ ENCODING also consists of 2 parts:

1) Encode $\gamma(|B(x)|)$ (GAMMA-ENCODING)
2) Encode $B(x)$.

} NO BITS shared!

DECODING:

1) First decode $\gamma(|B(x)|)$,
2) Then $B(x)$ ⇒ Get value of x in BINARY.

$$\delta(x) = \underbrace{\gamma(|B(x)|)}_{1+2\cdot\log L} \cdot \underbrace{B(x)}_{L}$$

BE ATTENTIVE: $\gamma(|B(x)|)$

CARDINALITY i.e. # BITS NEED TO REPRESENT x

Ex: $\delta(5) = \gamma(|B(x)|) \cdot B(x)$

$= 011 \quad 101$

More space than γ-ENCODING?

Let's actually check!

$$|\delta(x)| = \underbrace{L}_{|B(x)|} + \underbrace{2\cdot\log_2 L + 1}_{\gamma(|B(x)|)} = |ENC(x)|$$

$$|\gamma(x)| = \log_2 x + 2\log_2\log_2 x + 1$$

$>$

And we wanted first: $\gamma(x) = 2\cdot\log_2 x$

⇒ δ-ENCODING > γ-ENCODING
letter than
when x is large!

OPTIMAL DISTRIBUTION for δ-ENCODING:

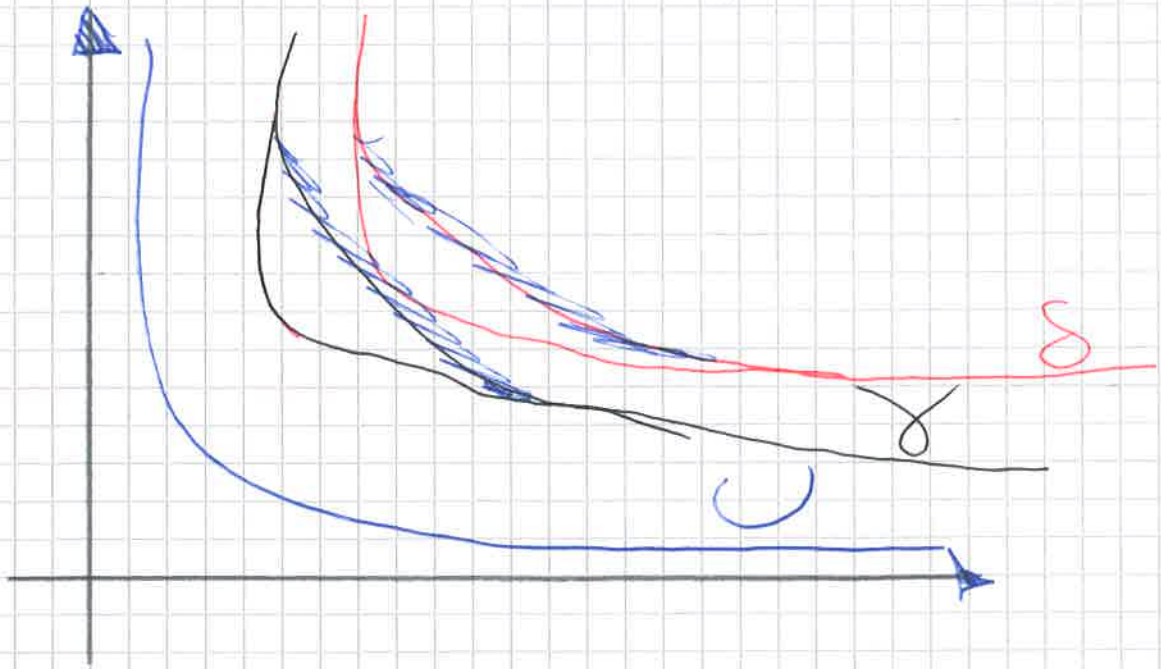
$|ENC(x)| = \log_2 \frac{1}{P(x)}$

⇒ STEPS!

$$P(x) = \frac{1}{2 \cdot x (\log x)^2}$$

⇒ $\lambda + O(\lambda)$ from length of OPTIMAL BINARY CODE.

⇒ γ and δ-CODES are UNIVERSAL and EFFICIENT when set is spread around 0.
— SLOW DECODING (BIT SHIFTS)



OPTIMAL DISTRIBUTION FOR δ-ENCODING.

$$|ENC(x)| = \log_2 \frac{1}{P(x)}$$

$$\log_2 x + 2\log_2 \log_2 x + 1 = \log_2 \frac{1}{P(x)}$$

$$\log_2 x + \log_2 (\log_2 x)^2 + \log_2 2 = \log_2 \frac{1}{P(x)}$$

$$\log_2 (2x) + \log_2 (\log_2 x)^2 = \log \frac{1}{P(x)}$$

$$\log (2x \cdot (\log x)^2) = \log \frac{1}{P(x)}$$

$$P(x) = \frac{1}{2x \cdot (\log x)^2}$$

RICE CODING:

Used when a set of integers are well-concentrated around a value $\neq 0$.

→ GOOD COMPRESSION RATIO & DECODING/ENCODING SPEED

PARAMETRIC ENCODING, based on parameter (K)

$$R_K(x) = \begin{cases} q = \left\lfloor \frac{(x-1)}{2^K} \right\rfloor & \text{QUOTIENT (represented in UNARY)} \\ r = x - 2^K \cdot q - 1 & \text{REMAINDER (K BITS)} \end{cases}$$

$r \in [0, 2^K - 1]$

$$R_K(x) = \boxed{U(q)} \; \boxed{r}$$

$q+1$ BITS (q "0s" + 1 "1") | K BITS

Q = QUOTIENT. Stored via $U(q)$ in a VARIABLE-LENGTH representation of $q+1$ BITS.

$q < 2^q$

r = REST. Stored via K BITS.

$r \in [0, 2^K]$,

$r < 2^K$

PARAMETER (k)'s CHOICE

Pick (k) in a way such that 2^k is concentrated at the mean of the elements of S.

(The more concentrated are elements around 2^k, the better the encoding)

Ex. $R_4(83)$, where $k=4$.

$$R_4(83) = \begin{cases} q = \left\lfloor \frac{83}{2^4} \right\rfloor = \left\lfloor \frac{83}{16} \right\rfloor = 5 \\ r = 83 - 2^4 \cdot 5 - 1 = 83 - 16 \cdot 5 - 1 = 2 \end{cases}$$

$U(q)$ | r in k bits

$R_4(83) =$ 000001 | 0010

$|R_k(x)| = q + 1 + k$ BITS

OPTIMAL DISTRIBUTION FOR RICE CODING

If values to be encoded follow a GEOMETRIC DISTRIBUTION with parameter (p) (i.e. $\mathbb{P}$), then we have:

$$P\{x\} = (1-p)^{x-1} \cdot p$$

If this is the OPT. DISTRIBUTION FOR RICE CODING

If GEOMETRIC DISTRIBUTION specified:

$$P[x] = (1-p)^{x-1} \cdot p$$

$$\Rightarrow k \geq 2^k \simeq \frac{\ln 2}{p} \simeq 0.69 \cdot \text{AVG}(S) \quad \text{OPTIMAL}$$

$\Rightarrow$ This encoding scheme can be used for encoding numbers not only at the beginning of the DISTRIBUTION!

SPACE: $\left\lfloor \frac{x-1}{2^k} + 1 + k \right\rfloor$ BITS.

P-for DELTA ENCODING:

§. EFFICIENT, FAST DECOMPRESSION & small size in the COMPRESSED OUTPUT, whenever:

REQUIREMENT:

S's values follow a GAUSSIAN DISTRIBUTION

2 PARAMETERS: (base, b)

IDEA: Force numbers repeating to be encoded with few bits

(Need a coded sequence of integers to identify them!

IDEA:

S =	(10)	12	15	16
BASE =	0	2	5	6
GAP-ENCODING =	0	2	3	3

Pick a BASE (encodes the first number in the sequence, even if not all numbers are non-negative).

b = 2 = # BITS to encode the resulting number

$$h_b = \begin{cases} h & \text{encoded with } b \text{ bits} \\ \text{escape} & \text{with } 2^b - 1 \end{cases}$$

if $h \in \{0, 1, \ldots, 2^b - 2\}$

if $h > 2^b - 2$

⇒ Put in a separate list to represent with k bits.

EXAMPLE: BASE = 1 ; b = 2

e.g. $1, 1 + 2^2 - 1$
$1, 3$

S =	(1)	3	13	29	31	37
S - BASE	0	2	12	28	30	36
GAP-ENCODING	0	2	(10)	(16)	2	6
bin:	00	10	11 (escape)	11	10	11
		01	(11)	(11)		

⇒ All values are encoded in fixed length — b bits ⇒

b + k bits ⇒ DECODED

NB: ESCAPED characters are encoded in a separate list using a fixed-size representation of (w) bits (WHOLE MEMORY WORD)

↳ They will be retrieved later on from a "special" data structure for ESCAPED CHARACTERS

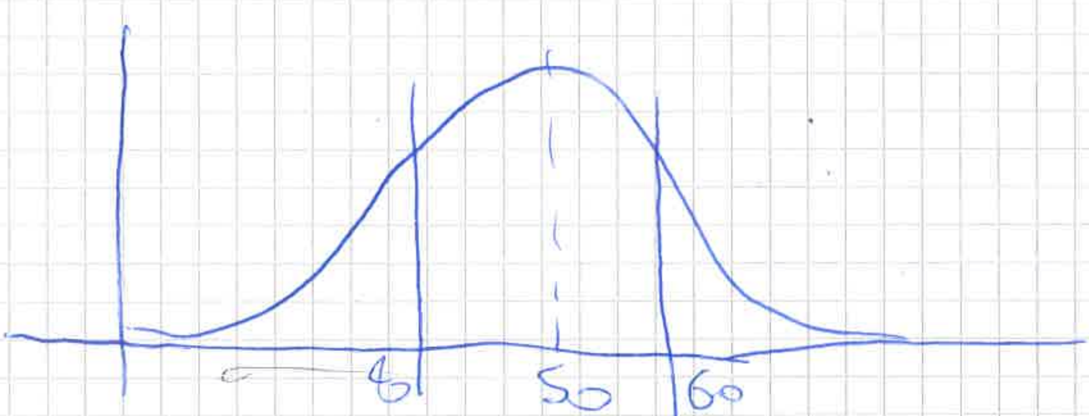
How to pick (b)?

(1) METHOD: Trade-off between SPACE WASTING (greater b) FASTER ENCODING / fewer exceptions and SPACE SAVING (more exceptions, smaller b)

(2) METHOD: Pick b in a way that: 90% of values in S are < 2^b.

APPLICATIONS:
Widely used in DATABASES!

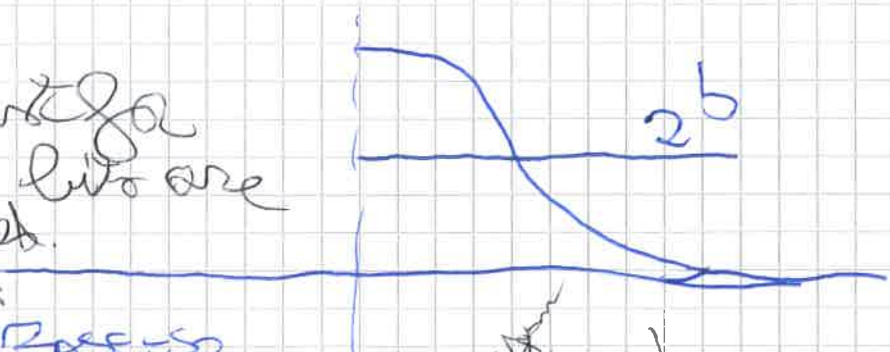
Ex:

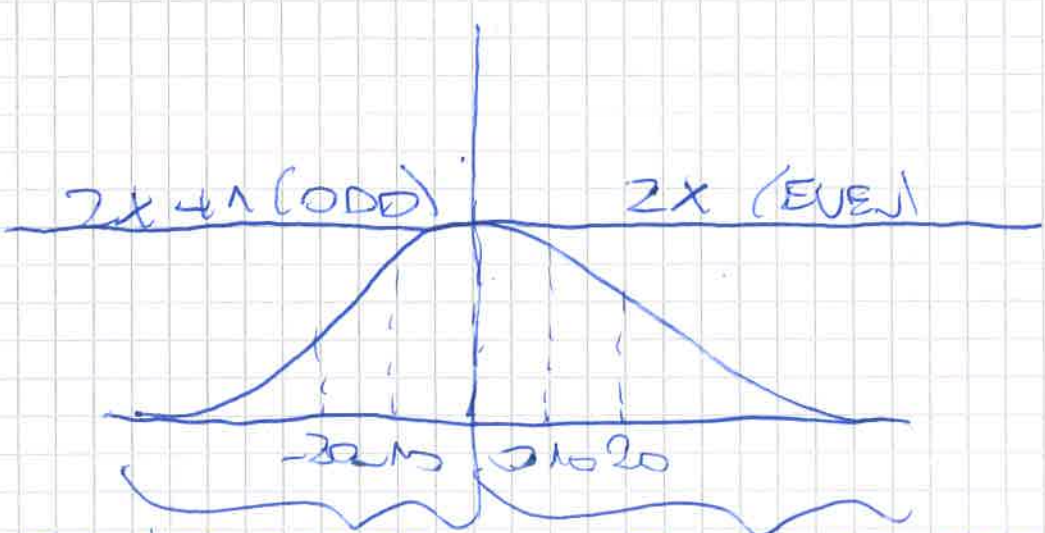


⇒ Keep an "array of signs":

ADVANTAGE

Extremely fast for decoding, as bits are word-aligned.
(can also avoid branch MISPREDICTIONS in some implementations)

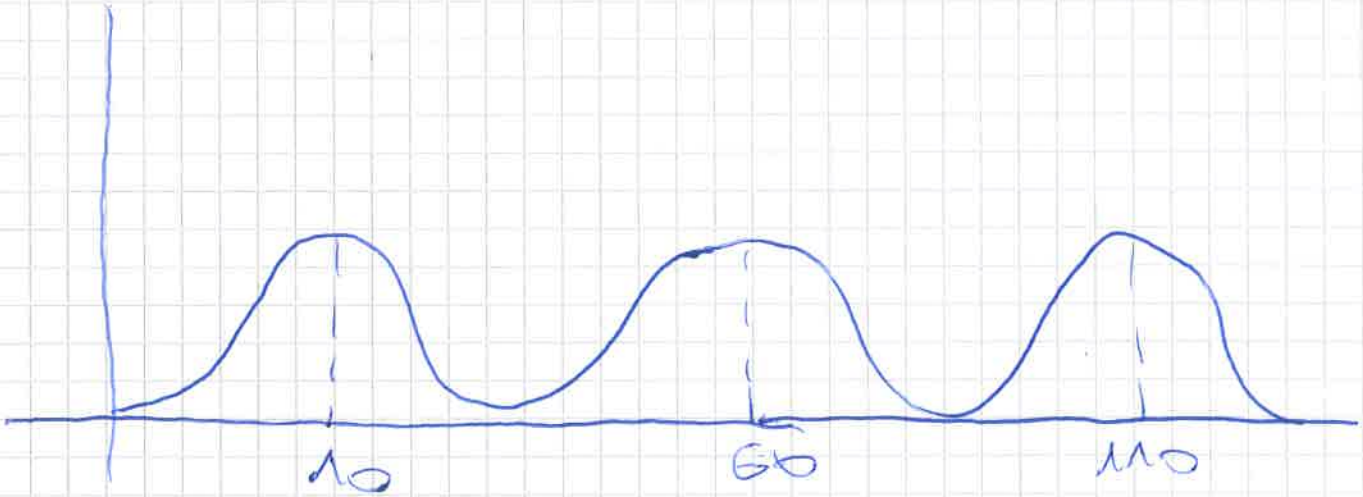




NEGATIVE NUMBERS
↓
encoded as ODD

POSITIVE NUMBERS
↓
encoded as EVEN

This doesn't work for a MULTI-MODAL DISTRIBUTION.



ENCODING EXERCISES (in class)

γ-CODING:

$S = 1, 5, 10, 13, 15, 20$

$\Delta(S) = 1\ 4\ 5\ 3\ 2\ 5$

2. GAP

ENCODING

$\gamma(\Delta(S)) = 1 \mid 00100 \mid 00100 \mid 01101 \mid 010$

δ-CODING: $\gamma(|B(x)|)$

$\delta(4) = \gamma(3)100 = 01100$

RICE-CODING:

$R_3(\Delta S)$, $k = 3$.

$R_k(x) = \underbrace{\text{UNARY}(q)}_{q+1 \text{ BITS}} \underbrace{r(3)}_{k \text{ bits}}$

q = QUOTIENT, stored in a UNARY string.

r = REST, stored with k BITS.

$\Delta S = 1\ 4\ 5\ 3\ 2\ 5$

$1\ 00 \mid 1100 \mid 1101 \mid 1011 \mid 1010$

$q+1$ bits, k bits

$q = \left\lfloor \frac{x-1}{2^k} \right\rfloor = 0$ $\quad 1101$

⇒ For all ENCODING TYPES ⇒ Say which distribution is optimal.

PFor Delta Coding.

S =	1	5	10	13	15	20
S-base =	-9	-5	0	3	5	10
POSITIVE =	19	11	0	6	10	20
BIN(x) =	111	111	000	110	111	111

ESCAPED SET = {19, 11, 10, 20}

base = 10

b = 3

NB: Numbers are made positive according to:

$$X = \begin{cases} -2X + 1 & \text{if } X < 0 \Rightarrow \text{ODD} \\ 2x & \text{if } X > 0 \Rightarrow \text{EVEN} \\ 0 & \text{if } x = 0 \end{cases}$$

In the BIN(x) representation, we make use of 3 bits ⇒ Hence:

- We represent x upto $2^b - 2$ (7)
- We escape $x > 2^b - 1$ with:

$\underbrace{111}_{b \text{ bits}} = 2^b - 1$ value for ESCAPING.

⇒ Escaped characters will be simply read from the list of escaped characters.

VARIABLE-BYTE CODE ~~DENSE CODE~~

Class of codes that trade speed with SUCCINCTNESS ⇒ One instantiation thereof is the VARIABLE-BYTE CODE, which uses a sequence of bytes to represent an integer X.

↳ SIGNIFICANT DECODING SPEED REMARKABLE

IDEA: Take a number X and represent it in BINARY ⇒ $B(X)$ ⇒ Partition $B(X)$ into groups of 7 bits, padded at the beginning by one BIT PADDING:
- 0 : STOP (last group)
- 1 : CONTINUE (non-last group)

DECODING: Keep scanning the BYTE SEQUENCE till we find a byte that is < 128 ($1000\,0000$, $2^7 = 128$)

(i.e.: Till we find a STOPPER GROUP)

EXAMPLE:

INPUT: X

$B(X)$ ⇒ PARTITION into groups of 7 bits (from the right)

(Add leading 0s or padding if needed)

EXAMPLE: ENCODE $X = 2^{16}$

$X = 2^{16}$ ⇒ $B(X) =$ 1 | 000 0010 | 000 0000

GROUP 0 | GROUP 1 | GROUP 2

GROUP 2: CONTINUE BIT → STOPPING POINT

1 000 0010 | 1 000 0000 | 0 000 0000 = ENCODING via VARIABLE-LENGTH BYTE.

PADDING

For ANY number x, we will have at least $2^7 = 128$ (1 Group)

SPACE = At least 8 bits (7 for representation + 1 for padding)

On average, 7 bits are wasted (given text x has 32 bits)

Remember:
1 byte = 8 bits
It can represent $2^8 = 256$

SPACE in BYTES = $\frac{|B(x)|}{7}$ (because one byte takes 8 bits 2 1 bit is used for PADDING)

SPACE in BITS = $8 \cdot \frac{|B(x)|}{7}$

NB: Drawback: When encoding very small digits, we waste bits & losing 0s for the representation!

Ex: 5

1 | 000 | 0101

WASTED

OPTIMAL DISTRIBUTION FOR VARIABLE BYTE CODE:

$$\underbrace{|ENC(x)|}_{\text{VARIABLE BYTE CODE}} = \log_2\left(\frac{1}{P(x)}\right)$$

$$\frac{8 \cdot |B(x)|}{7} = \log_2\left(\frac{1}{P(x)}\right)$$

$$8 \cdot |B(x)| = 7 \cdot \log_2\left(\frac{1}{P(x)}\right)$$

$$8 \cdot \log_2(x) = 7 \cdot \log_2\left(\frac{1}{P(x)}\right)$$

$$\log_2(x^8) = \log_2\left(\frac{1}{P(x)}\right)^7$$

$$x^8 = \left(\frac{1}{P(x)}\right)^7$$

$$x^8 = \frac{1^7}{(P(x))^7}$$

$$(P(x))^7 = \frac{1}{x^8}$$

$$P(x) = \sqrt[7]{\frac{1}{x^8}}$$

ADVANTAGE of VARIABLE-BYTE CODING:

GROUP-BASED decoding / BYTE-by-BYTE)
↳ Very fast!

Based on this bit, perform one of the 2 actions!

≥ 128
↓↓
(1) CONTINUE DECODING the next group

< 128
↓↓
(2) STOP DECODING the next group.

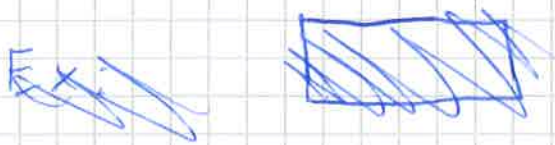
"CONTINUERS CONFIGURATIONS"
$|C| = C = 2^7 = 128$

"STOPPERS CONFIGURATIONS"
$|S| = S = 2^7 = 128$

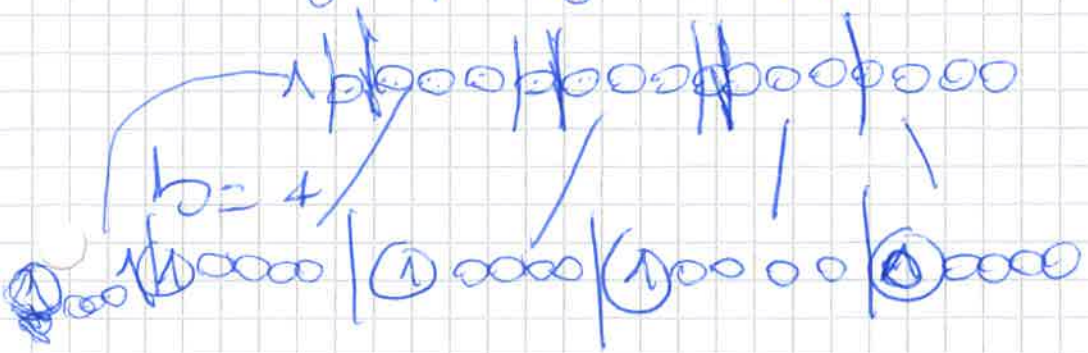
S-C-DENSE CODES:

$$S + C = 256 = 2^8$$ = Overall # possible configurations.

However, can we change the # BITS used to encode the GROUPS? (No longer just 7)



Ex: Do not expand into groups of 7 bits, but into groups of b bits.



⇒ For a GENERIC group size we would have:

2^b ≤ $(S)+(C) = 2^b$ ⇒ Based on the word's size!

#STUFFERS' CONFIGURATIONS in (b) bits.

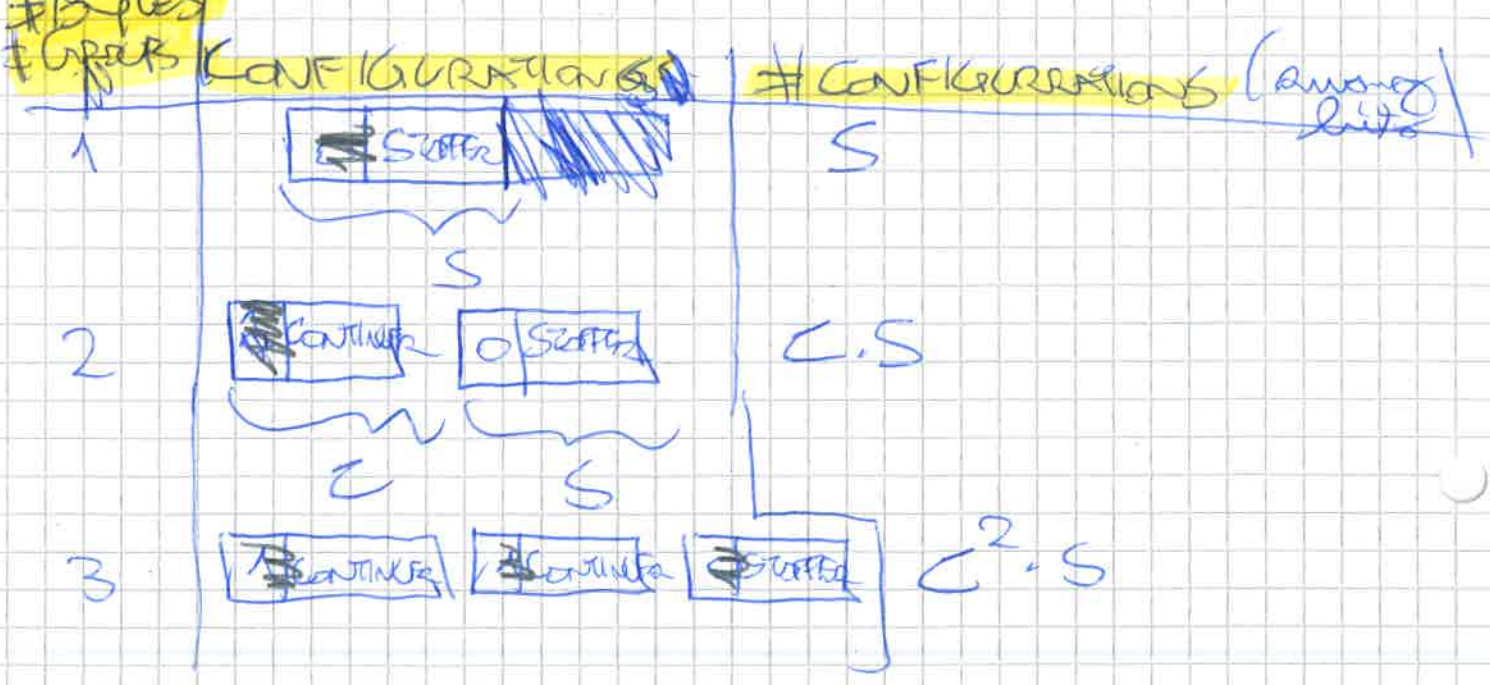
#CONTAINERS' CONFIGURATIONS in (b) bits.

⇒ The value to pick for (S) and (C) depends on the DISTRIBUTION of integers to be sorted.

How do we count CONFIGURATIONS?

CONFIGURATIONS' COMBINATIONS:

(i.e.: relative valid COMBINATIONS of configurations)



For n groups/layers: ⇒ n layers can encode $C^{n-1} \cdot S$ integers.

$$\#\text{CONFIGURATIONS} = C^{n-1} \cdot S$$

with n groups

Ex: $b=3 \Rightarrow \#\text{CONFIGURATIONS} = 2^b = S + C$

$2^3 = S + C \Rightarrow$ We can have encode values from $[0, 15]$

$3+1$ $S+1$ $1+1$

$S=6 \Rightarrow$ Encode values up to 5 with it.

VALUES:	$S=C=4$	$S=6, C=2$
0	000	000
1	001	001
2	010	010
3	011	011
4	100	100
5	101	101
6	110	110 000
7	100 011	110 001
8	100 000	110 010
9	100 001	110 011
10	100 010	110 100
11	100 011	110 101
12	110 000	111 000
13	110 001	111 001
14	110 010	111 010
15	110 011	111 011

STUFF 2 GROUPS

4 VALUES ENCODED

These bits not for config. 2.

100

[0-3] so ... 4

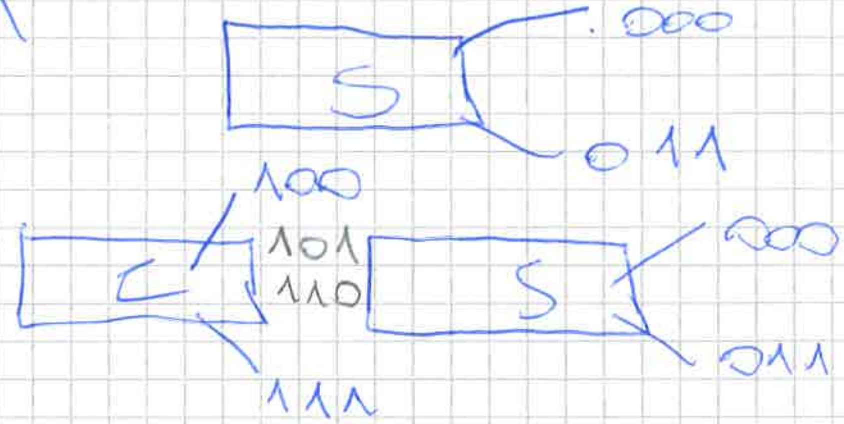
Difference between (1) S = 4; C = 4 and
(2) S = 6; C = 2

$b = 3 \Rightarrow 2^b = 8$

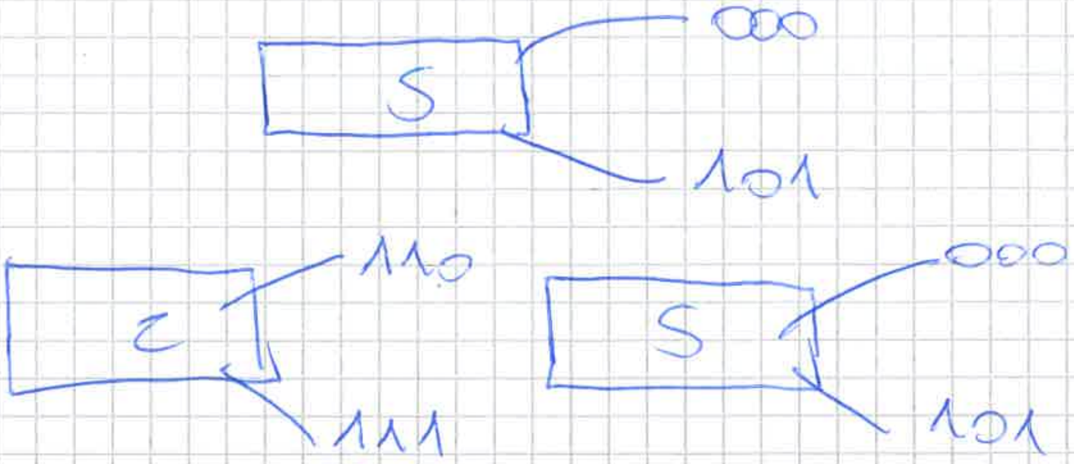
In both cases,
we use 2 words of 3 bits (6 bits) to encode all the integers.

In case (1): S = 4; C = 4, we encode the first four values with ONE word.

(1)



In case (2): S = 6; C = 2, we encode the first 6 values with ONE word, and the following ones with 2 values.



⇒ This can be more advantageous, based on the SKEWNESS in the distribution of {1, 15}.

S is small (stopper is small)

(S, C) DENSE CODE - CLASS EXERCISE

S = 1; C = 3 (1, 3) - CODE

⇒ S + C = 4 = 2^2 [Must be a Power of 2]

2 BITS per group.

CONFIGURATIONS

00 → STOPPER

01
10
11 } CONTINUERS

• CONSTRUCT THE FIRST 10 CODEWORDS

BLOCKS USED	RANGE & COMBINATIONS	INDEX	ENCODING
S=1 [S]	1 [0]	0	00
[C] [S] S=1 C=3	*3 [1-3]	1	01 00
		2	10 00
		3	11 00
[C] [C] [S]	9 [4-12]	4	01 01 00
		5	01 10 00
		6	01 11 00
		7	10 01 00
		8	10 10 00
		9	10 11 00
		10	11 01 00
		11	11 10 00
		12	11 11 00
[C] [C] [C] [S]	27 [13-39]	13	01 01 01 00
		14	01 01 10 00
		15	01 01 11 00

$C^2 \cdot S$

$C^3 \cdot S$

We keep going down as long as:

$MIN(n) < i$, where i is the starting position of α.

$\Rightarrow$ At the end, we return

$\langle i - m, |\alpha|, c \rangle$

$MIN(n)$, where n is the last value we visited at.

EX: How many bits do I need to represent the number 15?

$\Rightarrow$ Need 3 containers $\wedge$ topper

$\Downarrow$

3 bits needed.

#CONFIGS:

S=C=4	C=2; S=6	#CONFIGS
4	6	S
16	12	C S
64	24	C C S

111 ^ 101 = 6

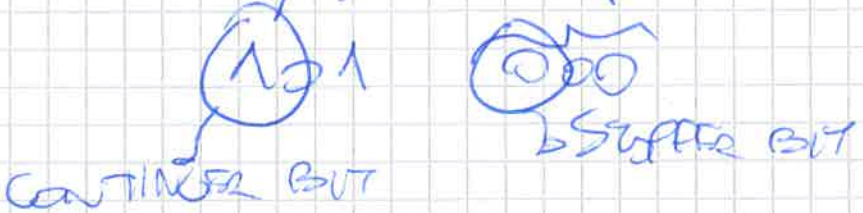
⇒ S and C can be chosen by minimizing a CONVEX FUNCTION.

APPROACH TO ENCODE a number:

C=S=4; b=3

X=8

#B(x) = 0001 | 000



C=2; S=6

X=13

B(x) = 00 1 | 1 0 1

INTERPOLATIVE CODE:

WHEN?

Integer-encoding technique ideally used when there is a CLUSTERED occurrence of integers [in small ranges] (EX: Store Posting lists → ZIP codes in search engines)

INPUT: Increasing sequence $S = (s_1, \dots, s_n)$

$$s_{i-1} < s_i < s_{i+1}$$

PRECEDING ELEMENTS | # FOLLOWING ELEMENTS to the MAX.

EX:

$S =$	(1)	2	3	5	7	[9]	11	15	18	19	20	(21)
OFFSET =	1	2	3	4	5	6	7	8	9	10	11	12

We exploit the fact that 21 is the max. number and that the numbers are INCREASING & DISTINCT.

IDEA: We want to encode [9] w.r.t. its lower bound & its upper bound.

UPPER BOUND = MAX − # FOLLOWING ELEMENTS to the MAX.

(Assuming S_{i+1} increases by 1 at every step till the end)

= 21 − 6

= 15

LOWER BOUND = # ELEMENTS PRECEDING to the MIN

= 6.

CONFIGURATIONS = 9 − 6 = 3

$\#BITS = \log_2(15 - 6 + 1) = \lceil \log_2 10 \rceil = 4$

All possibilities of offset (upper bound / lower bound)

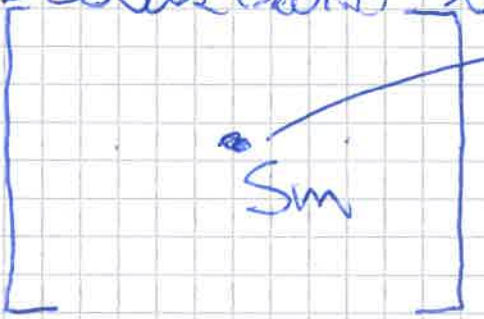
$\#BITS = \lceil \log_2(upper - lower + 1) \rceil$

↳ Very efficient encoding wrt. space.

IDEA: Each number in S is encoded with as few bits as possible.

↳ We here obtain an IMPLICIT ENCODING when considering the sequence of increasing numbers.

low = LOWER BOUND hi = UPPER BOUND



$low \leq X \leq hi$

BINARY CODE: $\langle X, low, hi \rangle$

We consider the BINARY ENCODING of $X - low$ in $\lceil \log_2(hi - low + 1) \rceil$ bits with

WHAT: $X - low$ is encoded in BINARY.

SPACE: $\lceil \log_2(hi - low + 1) \rceil$ (per number) $= n$

$S = S_1, S_2, \ldots, S_m, \ldots, S_{n-1}, S_n$

RECURSIVE ENCODING PROCEDURE:

Let's consider our INCREASING & DISTINCT sequence S:

S = 1 2 3 5 7 9 11 15 18 19 20 21
O = 1 2 3 4 5 6 7 8 9 10 11 12

At each RECURSIVE STEP, we know:

- l = left-index of $S'_{l,r}$ ⇒ low = min in $S'_{l,r}$
- r = right-index of $S'_{l,r}$ ⇒ hi = max in $S'_{l,r}$
- n = length of $S'_{l,r}$.

Where $S'_{l,r}$ is the sequence within a RECURSIVE CALL.

INITIALIZATION:

$n = |S'_{1,1}| = 1$; low = S'_1 ; hi = $S'_{n,r} = n$

Store in the COMPRESSED file for decoding the header to decode them.

MAINTENANCE:

In a recursive call, we operate similarly to BINARY SEARCH. Namely:

$$m = \left\lfloor \frac{l+r}{2} \right\rfloor$$

↳ Allows us to find S'_m

INVARIANT: In each recursive call, we have:

$\forall s'_i \in S'_{l,r}$:
- $s'_i > low$
- $s'_i < hi$

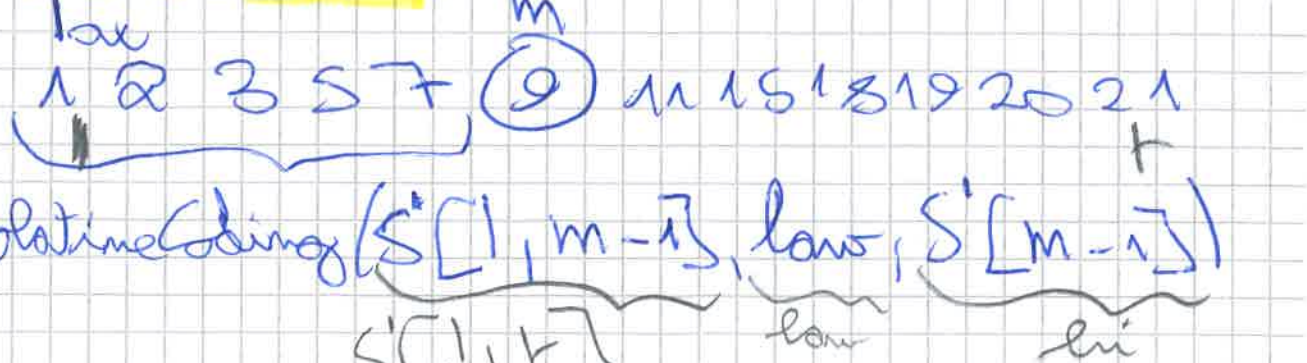
RECURSION'S IDEA:

We must encode the whole sequence (S) via a Binary-Encoding Procedure. How?

InterpolativeCoding(S'[1,r], low, hi):
  if r < l: // we have reached the end of a string
    return ϕ
  end if
  if l == r, then:
    return BinaryCode(S'[l], low, hi)   (S'[l] = low)
  end if
  compute m = ⌊(l+r)/2⌋

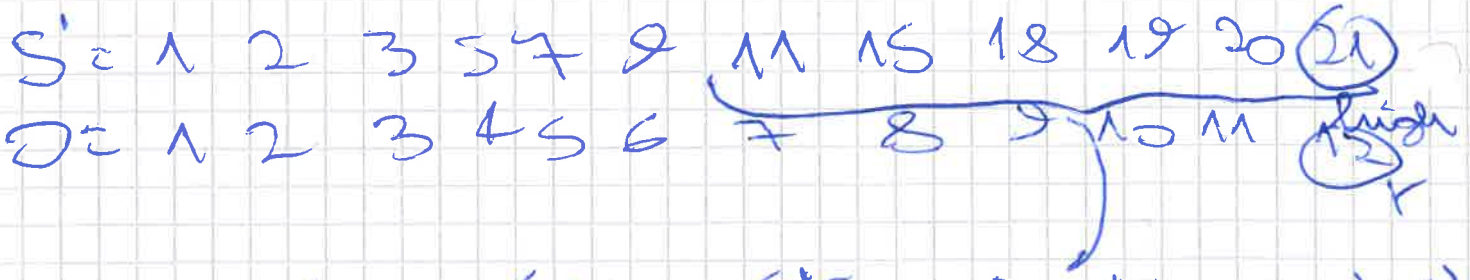
~~Compute A1 = BinaryCode(S'[m], low+m-1, hi-r+m)~~
// encode S'[m]
A1 = BinaryCode(S'[m], low+m-l, hi-r+m)
A2 = InterpolativeCoding(S'[l, m-1], low, S'[m]-1)
// go left of S'[m]
A3 = InterpolativeCoding(S'[m+1, r], S'[m]+1, hi)
// go right of S'[m]
return concatenation of A1·A2·A3;

VISUALIZED: LEFT.



InterpolativeCoding(S'[l, m-1], low, S'[m-1])
S'[l, r]

VISUALIZED: RIGHT



Interpolative coding: $S'[m-1, l]$, $S'[m+1, r]$, hi

$S'[l, r]$ low hi

NB: Low and High are VALUES!

1 and 1- are POSITIONS!

The encoding of an integer S'_i is not FIXED, but it depends on the distribution of fewer integers are made (VARIABLE ENCODING & ADAPTIVE)

& Low!

IY

NOT prefix-free.

⇒ EXPLOIT DISTINCTNESS & INCREASINGNESS of the elements in the sequence.

INTERPOLATIVE CODING VS SHANNON'S THEOREM

In certain cases, we are even able to do better than ~~dictated by~~ what SHANNON'S THEOREM dictates.

⇒ NO BITS used for encoding runs!

Ex: 1, 2, ..., 8

After emitting the first tuple, interval decoder knows

EXAMPLE. InterpolativeCoding(S[1,12], 1, 21)

S = 1 2 3 5 7 9 11 15 18 19 20 21

$m = \lfloor \frac{1+12}{2} \rfloor = 6$

A1 = BinaryCode(9, 1+6-1, 21-12+6)

BinaryCode(9, 6, 15) ⇒ 9-6 = 3 BITS

LEFT:

InterpolativeCoding(S[1,5], 1, 7)

1 2 3 5 7

$m = \lfloor \frac{1+7}{2} \rfloor = 3$

BinaryCode(3, 1+3-1, ..)

11

0

InterpolativeCoding(S[1,2], 1, 2)

$m = \lfloor \frac{1+2}{2} \rfloor = 1$

BinaryCode(

RIGHT:

InterpolativeCoding(S[7,12], 11, 21)

11 15 18 19 20 21

InterpolativeCoding(S[4,5], 5, 7)

How to encode? ⇒ BINARY ENCODING

low = 1, hi = 20

$S' = \{1, 2, 3, 5, 7, 9, 11, 15, 18, 19, 20\}$

$0 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$

Notice that: m on the LEFT, LOWER BOUND

LEFT: $S'[m] \geq low + m - 1$

$\geq 1 + 6 - 1$

We have in fact in $S'_{1,r}$:

$m-1$ distinct values in the left-hand side and smallest one is $\geq low$.

RIGHT: UPPER BOUND

$S'[m] \leq hi - (r - m)$

$S'[m] \leq hi - r + m$

⇒ We hence DO NOT encode $S'[m]$, but $S'[m] - (low + m - 1)$ KNOWN LOWER BOUND.

Encode with: $\log_2(S'[m] - (low + m - 1))$ BITS

We hence know that:

~~$hi - r + m \leq S'[m] \leq low + m - 1$~~

$low + m - 1 \leq S'[m] \leq hi - r + m$

DRAWBACK: To decode one element, need to know its LOWEST element.

INTERPOLATIVE CODING - CLASS EXERCISE.

$$S = [2, 5, 8, 10, 11, 12, 13]$$
$$\quad\ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7$$

$l = 1;\ r = 7;\ (n = 7);\ low = 2;\ hi = 13$

$m = \frac{l + r}{2} = 4 \Rightarrow S[m] = 10$

Binary Encode $(low + m - l;\ hi + r + m)$

Binary Encode $(S_m, 10) \Rightarrow S[m] - S_m = (5)$

in the #BITS needed to represent the Range!

$\Rightarrow$ 5 is represented in (3) bits.

RIGHT $\Rightarrow$ EMIT: 101

- LEFT of $S[m]$

$l = 5;\ r = 7;\ low = 11;\ hi = 13;\ n = 3$

~~Binary Encode~~

$m = \frac{l + r}{2} = 6$

Binary Encode $(12, 12)$

$\Rightarrow$ Encode $12 - 12 = 0 \Rightarrow$ No bits are emitted

- LEFT of $S[m]$.

~~$\Rightarrow$ Binary Encode~~

$l = 1;\ r = 3;\ low = 2;\ hi = 8;\ n = 3$

$m = \frac{l + r}{2} = 2 \Rightarrow S[m] = 5$

Binary Encode $(3, 7) \Rightarrow$ Encode $S[m] - 3 \Rightarrow 5 - 3 = 2$

Encosta (2) muito 3 lixos

EMISSÃO 10

ELIAS-FANO ENCODING:

≠ INTERPOLATIVE CODING!

WHEN? WHAT?

Code that does NOT depend on the distribution of integers to be encoded, allowing for INDEXING through compressed data structure.

WHEN? RANDOM ACCESS

→ Efficiently ACCESS the encoded integers in a compressed way

(Ex: Usage: Store inverted lists in search engines & adjacency lists of large graphs).

BAD: Space is a CONCERN for integers once ordered

REQUIREMENTS:

(S) is a monotonically increasing SEQUENCE (like for INTERPOLATIVE CODING) of DISTINCT numbers.

$$S_{i-1} < S_i < S_{i+1}$$

Ex: 1 3 5 7 12 15 18 20

Each INTEGER is represented in BINARY:

$b = \lceil \log_2 \rceil$ = # bits

INDEX(#)	bin(#)
1	
3	
5	
12	

~~Let's~~ FUNCTIONING EXPLAINED.

Let's take $S = \{1, 3, 9, 12\}$

Each integer is represented in BINARY as:

#BITS per INTEGER $= b = \lceil \log_2 U \rceil$, where

$U = \max S_i$ (i.e: max. integer in S)

$b = \lceil \log_2 12 \rceil = 4$

S_i	BIN(S_i)
1	0001
3	0011
9	1001
12	1100

$b = \lceil \log_2 U \rceil$

$l + w = b$

$w = b - l$ or $l = b - w$

Now let's consider a positive integer l and a positive integer w to PARTITION the BINARY REPRESENTATION into 2 BLOCKS.

$l = \lceil \log_2 \frac{U}{n} \rceil = 2$ = #BITS less significant

$n = |S|$

$w = \log_2 (U+n) - l = 4 - 2 = 2$ = #BITS most significant

We call the two representations respectively:

- ~~H~~ $L(S_i)$: Block of the l least significant bits (lowest ones)
- $H(S_i)$: Block of the $b-l$ most significant bits.

And we will surely have that:

$$|H(S_i)| + |L(S_i)| = b \text{ bits}$$

$|L(S_i)| = l \cdot n$ BITS $= n \log_2 \binom{w}{n}$

$|H(S_i)| = \underbrace{n}_{\text{OVERALL \# ONES}} + \underbrace{2^w}_{\text{\# ZEROES}}$ BITS $\Rightarrow$ SPACE for MOST SIGNIFICANT BITS

EXAMPLE:

S_i	$B(S_i)$
00	0001
01	0011
10	0101
11	0002

$H(S_i) = 10$

$L(S_i) = 01 \mid 11 \mid 01 \mid 00$

~~$H(S_i) = 00\ 00\ 10\ 11$~~

We group the most-significant BITS into BUCKETS, then represent each bucket by the COUNT of groups in that BUCKET IN UNARY.

For H, we identify the following BUCKETS

BUCKETS:	00	01	10	11
GROUP COUNT	2	0	1	1
GROUP COUNT (UNARY)	110	0	10	10

$= H(S_i)$

$H(s_i)$ has 2^w possible CONFIGURATIONS.

SPACE $= |L(s_i)| + |H(s_i)|$

$= n \cdot |L(s_i)| = n \cdot \left(\log_2 \frac{U}{n}\right)$

The FINAL ENCODING is given by:

$$|H(s_i) \odot L(s_i)|$$

(CONCATENATED)

$= 110\ 010\ 10 \mid 01\ 11\ 01\ 00$

$|L(s_i)| = n \cdot \left(\log_2 \frac{U}{n}\right)$ = SPACE of each INTEGER (log significant bits)

$|H(s_i)| = n + 2^w$ = All possible combinations (BUCKETS) with w bits.

→ IN UNARY!

0-BITS, 1-BITS

SPACE $\approx |L(s_i)| + |H(s_i)|$

$= n \cdot \left(\log_2 \frac{U}{n}\right) + n + 2^w$

Where $w = \log_2(U+1) - \log_2 \frac{U}{n}$ (l)

$\approx \log_2 U - \log_2 \frac{U}{n}$

$\approx \log_2 \frac{U \cdot n}{U}$

$\Rightarrow w = \log_2 n$

=> Let's plug in Elias value & we:

$$\text{SPACE (in bits)} \approx n \cdot \left(\log_2 \frac{U}{n} \right) + n + 2\log_2 n$$

$$\approx n \cdot \left(\log_2 \frac{U}{n} \right) + n + n$$

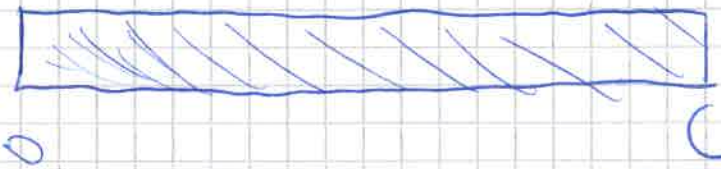
$$= n \cdot \left(\log_2 \frac{U}{n} \right) + 2n$$

$$= n \cdot \left[\log_2 \frac{U}{n} + 2 \right]$$

Regardless of the DISTRIBUTION! 1/2 BITS per integer!

Very close to an OPTIMAL SPACE!

LOWER BOUND & SPACE-OPTIMAL:



Let's consider the possible combinations of n elements into a space of size U.

$$\#\text{COMBINATIONS} = \binom{U}{n}$$

SPACE FOR all COMBINATIONS

$$\# = \log_2 \binom{U}{n} \approx n \cdot \log_2 \left(\frac{U}{n} \right)$$

(STIRLING'S FORMULA)

$$\Rightarrow \underbrace{n \cdot \log_2 \left(\frac{U}{n} \right)}_{\text{OPTIMAL}} \leq \underbrace{n \cdot \left(\log_2 \frac{U}{n} \right) + 2n}_{\text{ELIAS-FANO}}$$

LOWER BOUND! =>

=> Elias-Fano takes 2 bits more per integer (n) than the OPTIMAL LOWER BOUND (if integers are uniformly distributed.)

ELIAS-FANO

DECOMPRESSION: Easy & random access thanks to ZEROES.

CLASS EXERCISE

ELIAS - FANO CODING

$S = \{1, 5, 10, 13, 15, 20\}$

S_i	$B(S_i)$
1	0001
5	0010 1
10	0101 0
13	0110 1
15	0111 1
20	1010 1

$n = |S| = 6$

3 4 5
8 16 32

$l = \lceil \log_2 \frac{U}{n} \rceil = \lceil \log_2 \frac{20}{6} \rceil = \lceil \log_2 3.2 \rceil = 2$

$w = \log_2(U+1) - l = 5 - 2 = 3$

$l + w = 5$ ✓

$L(S_i) = 01 \mid 001 \mid 10 \mid 011 \mid 00 \mid 10$

BUCKETS: 000 001 010 011 100 101 110 111

GROUPS' COUNT: 1 1 1 2 1 1 0 0

$U(\text{GROUPS' COUNT})$: 10 10 10 110 10 10 0 0

$H(S_i)$

FINAL ENCODING

$\Rightarrow$ $H(S_i) \oplus L(S_i)$

$= 10\,10\,10\,110\,10\,10\,00 \cdot 01\,001\,10\,011\,00$

(42) # HUFFMAN CODING:

Part of the larger family of STATISTICAL CODING ⇒ It involves 2 phases:

- MODELLING PHASE:

Compute statistical properties of input & build a model.

- CODING PHASE:

Actually compress the input sequence.

STATISTICAL CODING can be applied to:

- Sequences of symbols (TEXTS) drawn from an alphabet $|\Sigma|$ (of finite size)
  ↳ TEXT ENCODING.
- Sequences of INTEGERS (NUMBERS), drawn from an alphabet $|\Sigma|$ (of infinite size)
  ↳ INTEGER CODING
- Genomes CODING

⇒ OPTIMAL ENCODING can be achieved for any distribution via HUFFMAN ENCODING (at the cost of storing the PREAMBLE (i.e. some overhead).

⇒ If an OPTIMAL distribution of the data to encode is possible PREFIX-FREE, then we should go for it to reduce the overhead and avoid complicated encoding.

How to HUFFMAN-ENCODE (i.e. functioning)

Recall the definition of entropy H:

$$H = \sum_{\theta} P\{\theta\} \cdot \log_2 \frac{1}{P\{\theta\}},$$

where θ is a symbol in Σ alphabet.

Code's length: (AVERAGE CODEWORD's LENGTH)

$$L_C = \sum_{\theta} P\{\theta\} \cdot len\_code(\theta)$$

$$\sum_{\theta \in \Sigma} P\{\theta\} = 1$$

SHANNON'S THEOREM:

If for every symbol we have a code, then a LOWER BOUND holds!

$$L_C \geq H \text{ bits, where.}$$

$$0 \leq H \leq \log_2 |\Sigma|$$

HUFFMAN FUNCTIONING: ENCODING

Huffman coding is based on a GREEDY LOCAL CHOICE, that eventually leads to an OPTIMAL situation ⇒ BUILD HUFFMAN TREE!

INPUT:

INITIALLY: ⇒ We have a set of symbols θ taken from Σ, each one with $P\{\theta\}$ of occurring ⇒ These symbols constitute the LEAVES of our tree.

Ex:

=> SYMBOLS & a corresponding PROB. DISTRIBUTION.

a	b	c	d	e	f
0.05	0.1	0.15	0.3	0.25	0.15

} LEAVES from CANDIDATE NODES

CANDIDATE NODES

REPEAT, at every step:

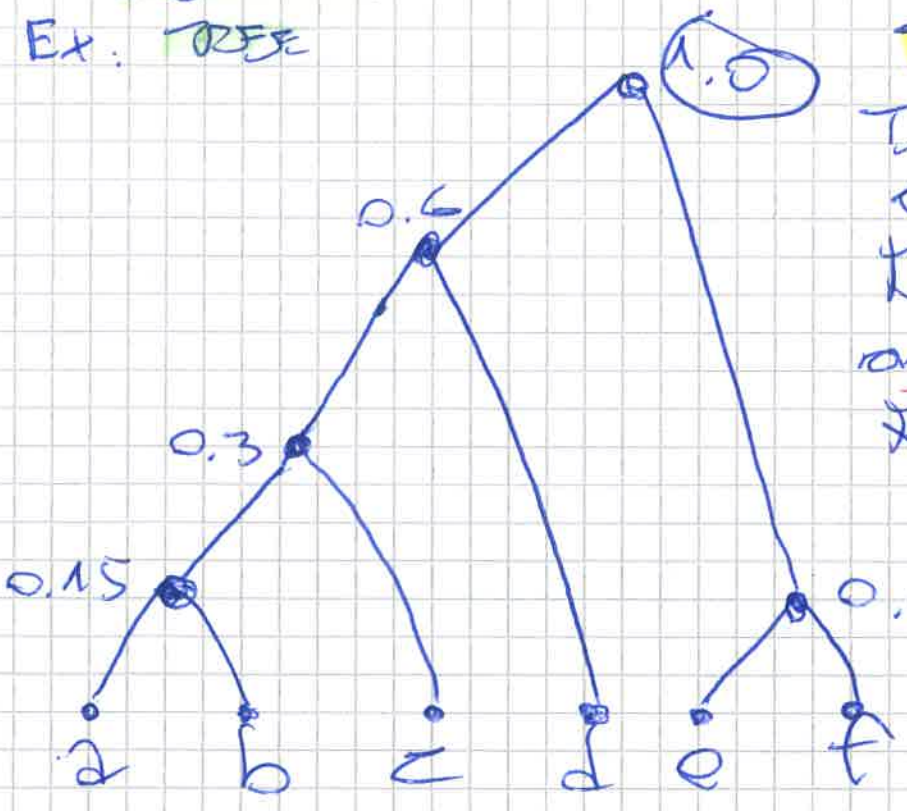
(1) Pick 2 leaves / nodes with the lowest PROBABILITY P[Θ] & merge them into a new node => add to CANDIDATE SET

(2) Sum their PROBABILITY P[Θ] into the newly created node & Remove merged nodes from the CANDIDATE SET

[NB: In case of a TIE among the sum of nodes' probability, just pick one pair => The tree would change, yet OPTIMALITY is preserved]

HUFFMAN

Ex: TREE



TERMINATION:

The algorithm continues until there is only one node (the root) with P[Θ] = 1.

=> How to get the HUFFMAN CODE based on the HUFFMAN TREE?

(2) Get the HUFFMAN CODE based on the HUFFMAN TREE. How? -> LABELED HUFFMAN TREE

We assign BINARY LABELS (0 or 1) to all the TREE EDGES of the HUFFMAN TREE.

↳ Generally:

↳ LEFT EDGE of an internal node => 0

↳ RIGHT EDGE of an internal node => 1

INTERNAL NODES in a HUFFMAN TREE $= n-1$

POSSIBLE HUFFMAN CODES in a HUFFMAN TREE $= 2^{|\Sigma|-1} = 2^{n-1}$

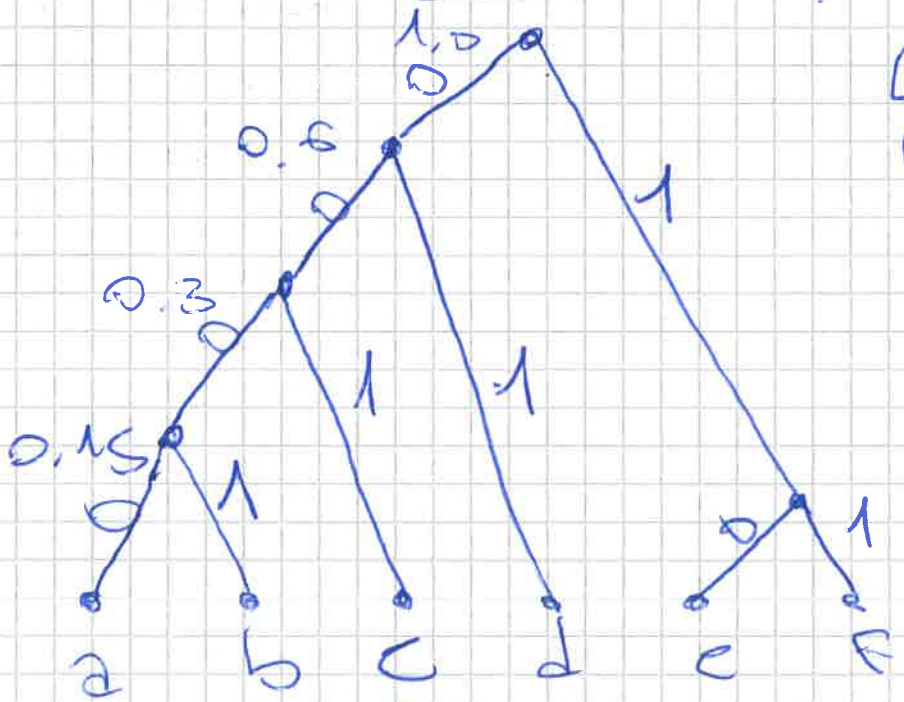
[2 CHOICES: 0 or 1, in each edge, for each internal node]

↳ Based on a LABELED HUFFMAN TREE, we can

(3) then obtain the HUFFMAN CODEWORD for a symbol Θ by:

- Taking the BINARY LABELS encountered on the downward path connecting the root to the leaf Θ.

Ex. Build LABELED HUFFMAN TREE & create a HUFFMAN CODEWORD.



LEFT EDGE => 0
RIGHT EDGE => 1

GOLDEN RULE of statistical data encoding:
The rarer a symbol, the more bits we use to encode it.
The more frequent a symbol, the fewer bits we use to encode it.

CODEWORD of θ = a = 0000
b = 0001
c = 001
d = 01
e = 10
f = 11

$L(\theta)$ = length of the HUFFMAN CODEWORD of θ

NB. We notice that making a different choice in picking symbols with equal probability has an impact on the MAX. codeword length which would impact on the size of the COMPRESSION / DECOMPRESSION BUFFER

So IDEA to minimize the MAXIMUM CODEWORD LENGTH, we consider the OLDEST nodes

⇒ OLDEST NODES = Leaves or internal nodes that have been merged farthest in the past than the other nodes in the candidate set.

⇒ Use 2 QUEUES:

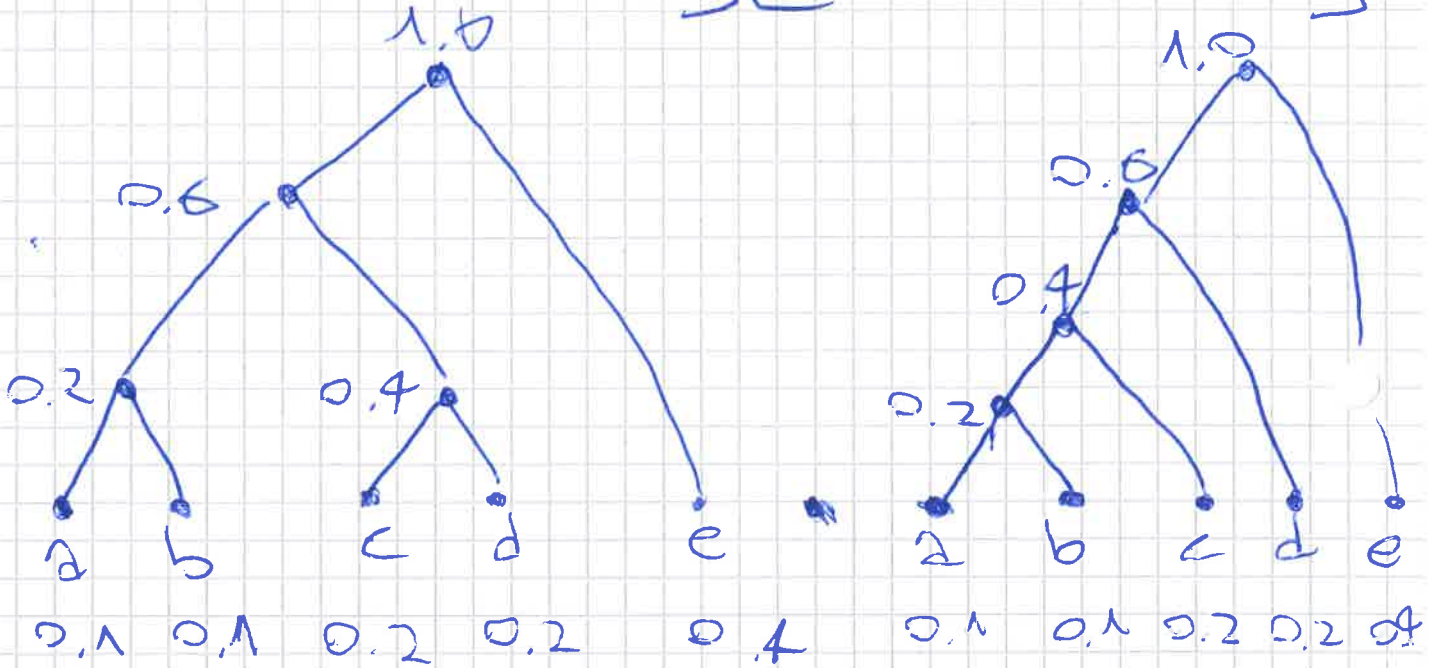
① QUEUE 1 containing symbols sorted by increasing probability

② QUEUE 2 containing internal nodes in the order they are created by the Huffman algorithm.

TIE:

[Node of QUEUE 1 ; then on QUEUE 2]

[APPROACH = QUEUE - head] [APPROACH - Random]



Huffman is NOT the only OPTIMAL encoding algorithm ⇒ We can have many HUFFMAN encoding algorithms that are OPTIMAL as well.

⇒ Since Huffman is OPTIMAL, then C must be optimal as well.

COMPRESSED FILE, following HUFFMAN'S ALGORITHM APPLICATION:

$\Theta(|\Sigma|)$ (HUFFMAN TREE)

Codewords of symbols of Θ & Σ.

AVG. HUFFMAN CODEWORD LENGTH:

L_H = Average Codeword length produced by Huffman

$$H \leq L_H \leq H+1 \Rightarrow \text{OPTIMAL}$$

lower & upper

$\Rightarrow$ The AVG. codeword length is hence bounded by the ENTROPY $H = \sum_{\Theta} P(\Theta) \cdot \log_2 \frac{1}{P(\Theta)}$

knowing the entropy H, we can balance this route ~ codeword's length

THEOREM – OPTIMALITY of HUFFMAN CODING:

If C is a HUFFMAN CODE, then L_C is the shortest possible AVG. length among all the PREFIX-FREE codes C', where:

$$L_C^H \leq L_{C'}$$ → codeword of any other prefix-free code C'

$$L_C^H = \sum_{\Theta \in \Sigma} P(\Theta) \cdot L(\Theta)$$

PROOF: We notice that a prefix-free code can be seen as a BINARY TREE.

$\Rightarrow$ We show the minimality of AVG. length path to prove HUFFMAN CODE's OPTIMALITY.

⇒ How to PROVE:

If the Theorem does not hold, then:

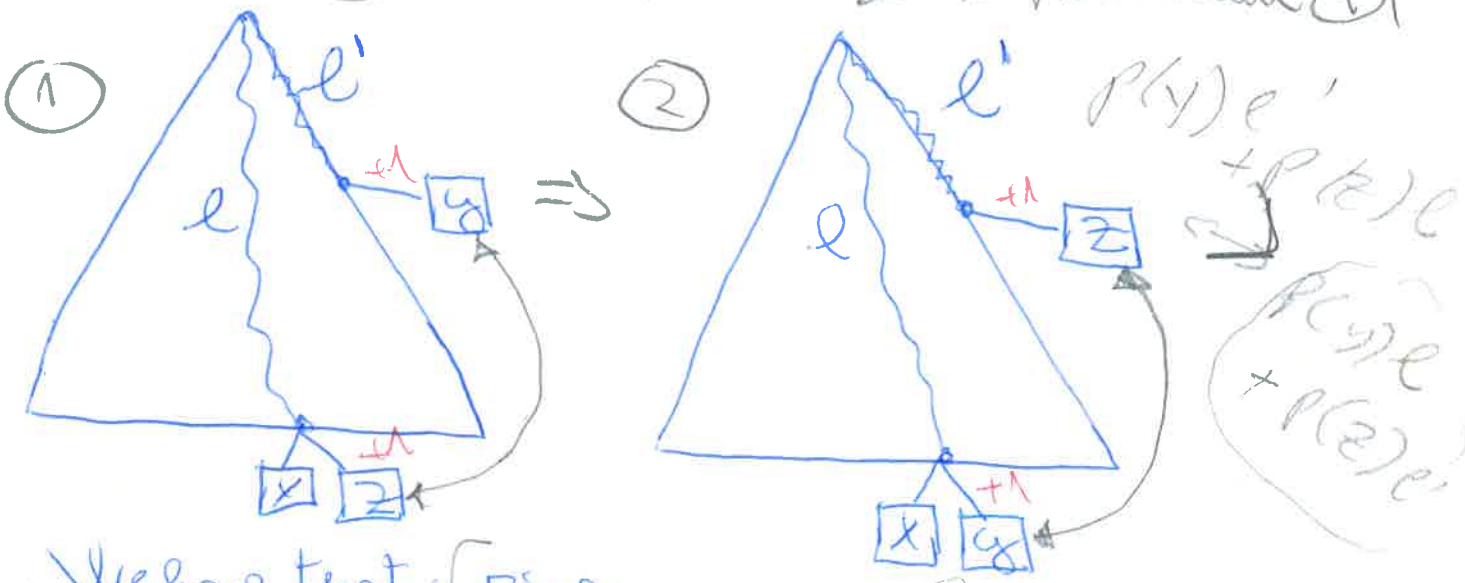
- A NOT minimum probability leaf (say (z)) occurs at the deepest level of the binary tree. (1) CONFIGURATION

- (2) can hence be swapped with a MIN. probability leaf (say (y)) and reduce the AVG. depth of the resulting binary tree.

(2) CONFIGURATION.

Proof THM ⇒ If CONFIGURATION (2) is less advantageous than CONFIGURATION (1).

(i.e: (2) has least average depth than (1))



We have that: $P\{z\} \geq P\{x\} \cdot P\{y\}$

(1) $l \cdot P\{x\} + (l+1) P\{z\} + (l'+1) \cdot P\{y\} + \dots$

(2) $(l+1) \cdot P\{x\} + (l+1) P\{y\} + (l'+1) \cdot P\{z\} + \dots$

(1) $l \cdot P\{z\} + P\{x\} + l' \cdot P\{y\} + P\{y\}$

(2) $l \cdot P\{y\} + P\{y\} + l' \cdot P\{z\} + P\{z\}$

~~COMPRESSED FILE following~~

$\Rightarrow$ We now show that indeed:

CONFIGURATION (1) $\rightarrow$ CONFIGURATION (2)

$$l \cdot P\{z\} + l'' P\{y\} \rightarrow l P\{y\} + l' P\{z\}$$

$$P\{z\}(l - l') \rightarrow P\{y\}(l - l')$$

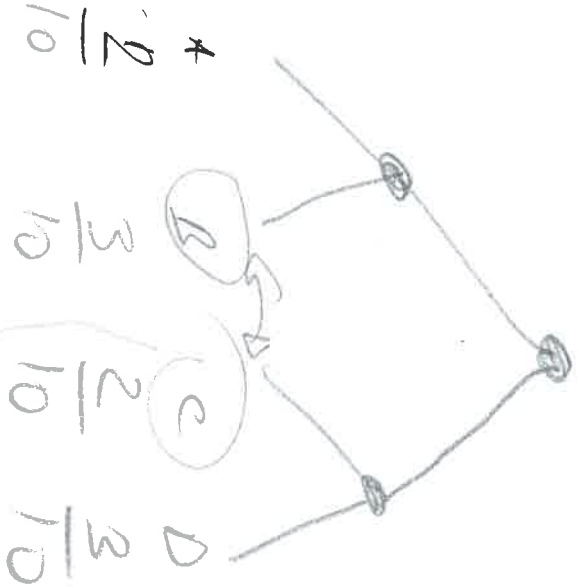
$\Rightarrow P\{z\} \rightarrow P\{y\}$ Q.E.D!

LEMMA 1:

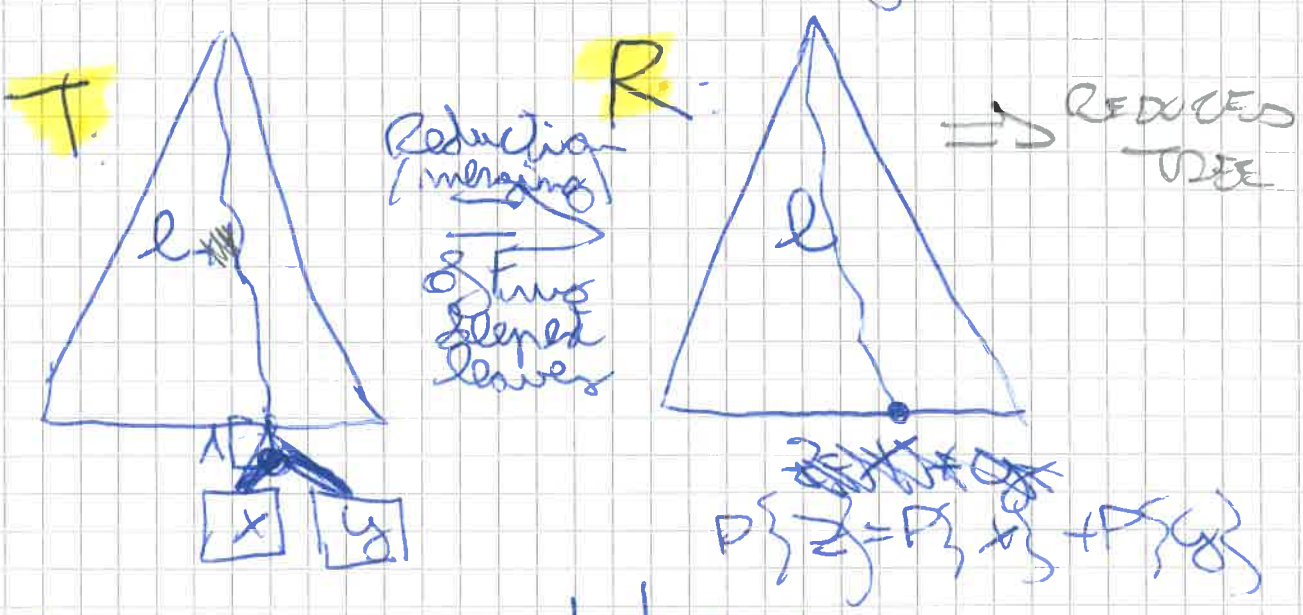
Let T be a BINARY TREE whose AVG. depth is minimum among the binary trees with $|\Sigma|$ leaves.

$\Rightarrow$ Then, the two leaves with MIN. probability will be at the greatest depth of T and will be children of the same parent node.

PROOF: Consider an alphabet Σ with n symbols, and (x) and (y) have the smallest PROB. (i.e. these are candidates with minimum P.)



The situation is hence the following:



L_T:

$$(l+1)(P\{x\}+P\{y\}) + \Delta$$

$$l\cdot(P\{x\}+P\{y\}) + (P\{x\}+P\{y\}) + \Delta$$

$$\underbrace{l\cdot(P\{x\}+P\{y\}) + (P\{x\}+P\{y\}) + \Delta}_{L_T}$$

L_R:

$$l(P\{x\}+P\{y\}) + \Delta$$

$$\underbrace{l\cdot(P\{x\}+P\{y\}) + \Delta}_{L_R}$$

$\Rightarrow L_T = L_R$ when ?. LEMMA 2:

$$\boxed{L_T = L_R + (P\{x\}+P\{y\})}$$

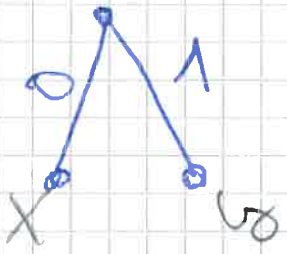
We now want to prove the OPTIMALITY of HUFFMAN CODE, by drawing that any other code, say Σ, would be longer than HUFFMAN CODING.

OPTIMALITY & HUFFMAN CODING:

$\forall \bar{C}, \boxed{L_{\bar{C}} \geq L_H}$

PROOF by INDUCTION on n, the #SYMBOLS in Σ.

- BASE CASE: Take n=2 symbols.



⇒ OBVIOUSLY OPTIMAL, as it assigns one single bit 0 to one symbol and one single bit 1 to the other symbol.

- INDUCTIVE STEP: Let's consider $n > 2$ symbols in Σ, and assume:

Assume:

HYPOTHESIS: Huffman code is OPTIMAL for $(n-1)$ symbols.

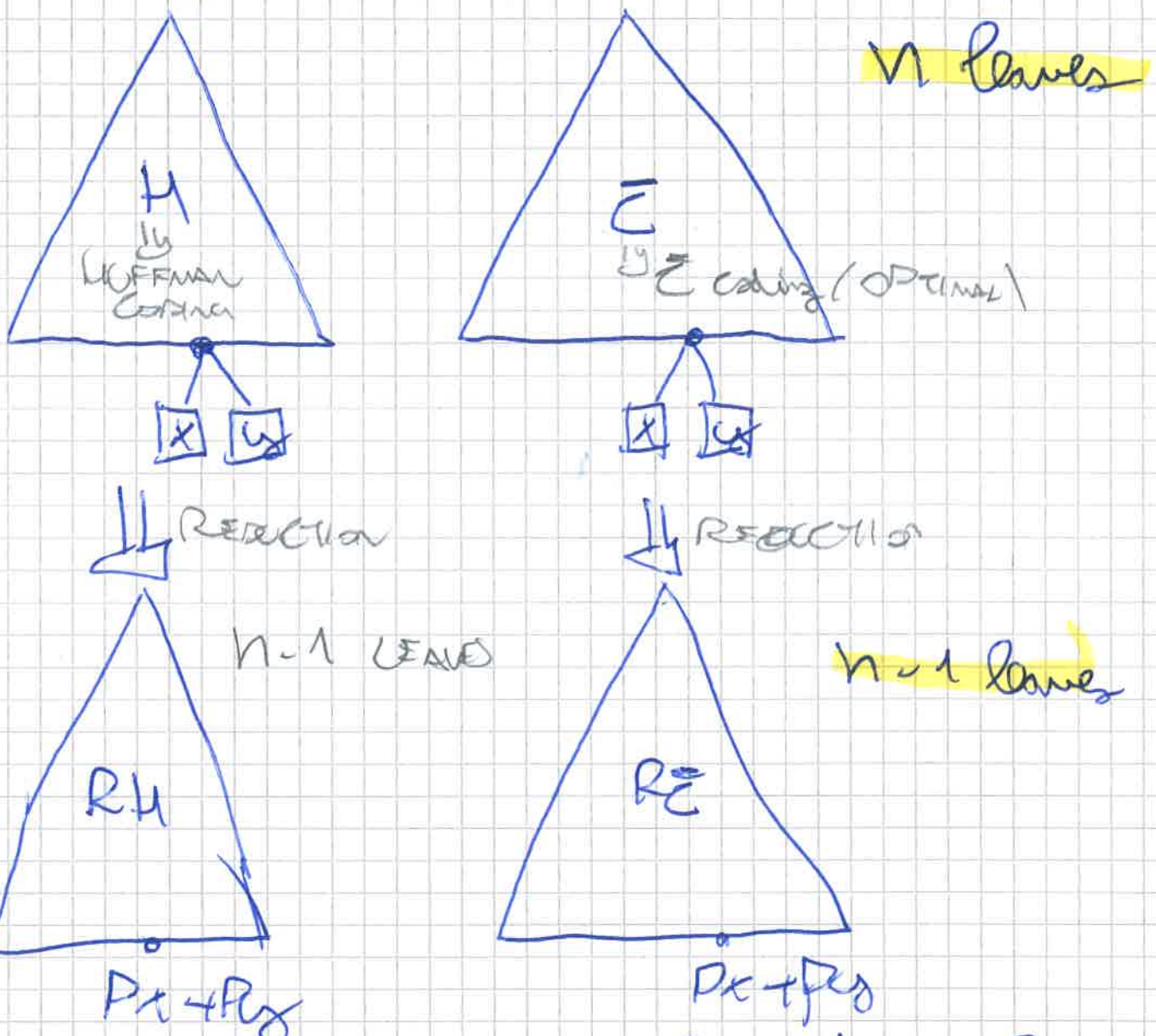
⇒ Take $|\Sigma| = n$ and let $\bar{C}$ be an OPTIMAL CODE for Σ and its underlying distribution.

We will want to show that:

$\boxed{L_{\bar{C}} = L_H}$ (i.e. Huffman is also OPTIMAL)

and this will have to hold for (n) symbols. like Code C

Let's now consider two BINARY TREES with (n) leaves.



Following a REDUCTION, RH and RT have respectively n-1 leaves.

⇒ Let's now apply our inductive hypothesis for RH and RT, which have n-1 leaves.

- By INDUCTIVE HYPOTHESIS, RH is OPTIMAL for the REDUCED alphabet (where x and y have been merged). Hence:

$$L_{RH} \leq L_{RT}$$

AVG. depth of an HUFFMAN CODE

AVG. depth of an OPTIMAL CODE after the REDUCTION

We can now apply LEMMA 2 over

$$L_{RH} \leq L_{R\bar{C}}$$

Where we know that by LEMMA 2:

$$L_T = L_C + (P\{x\} + P\{y\})$$

L_T: n leaves; L_C: n-1 leaves

Hence, in the present situation:

$$L_H = L_{RH} + (P\{x\} + P\{y\})$$

And by LEMMA 2 again:

$$L_{\bar{C}} = L_{R\bar{C}} + (P\{x\} + P\{y\})$$

$$\Rightarrow \begin{cases} L_{RH} = L_H - (P\{x\} + P\{y\}) \\ L_{R\bar{C}} = L_{\bar{C}} - (P\{x\} + P\{y\}) \end{cases}$$

SUBSTITUTING NOW we get:

$$\Rightarrow L_H - (P\{x\} + P\{y\}) \leq L_{\bar{C}} - (P\{x\} + P\{y\})$$

$$\Rightarrow L_H \leq L_C$$

Q.E.D.

We have therefore proved that HUFFMAN CODE

ISSUE in ENCODING WITH HUFFMAN.

We always pay some overhead to store the preamble, which contains our HUFFMAN TREE.

However, if our HUFFMAN TREE is small, the preamble is NEGLIGIBLE $\Rightarrow$ N.H overall

$$H \leq L_H \leq H + 1$$

1 extra bit per symbol.

If we have a very large H, 1 bit per symbol is negligible.
Otherwise, if we don't have a very large H, 1 bit can have an impact.
(We cannot encode a symbol with < 1 bit!)

$\Rightarrow$ HUFFMAN & SHANNON'S INTUITION: BLOCKING SCHEME $k \to \infty$

If you want to encode a very large text T, say of infinite size, you can chunk it into k-chunks of FINITE SIZE $|k|$.

$$T = \underbrace{\quad}_{k} \underbrace{\quad}_{k} \underbrace{\quad}_{k} \underbrace{\quad}_{k} \underbrace{\quad}_{k}$$

$\Rightarrow \Sigma' = \{$ blocks of k symbols $\}$

Σ': extended alphabet of size Σ^k, whose symbols are substrings of k-symbols.

⇒ If we use Huffman on the $-Blocks, the extra bit is lost!

⇒ The larger is the block, the smaller is the penalty I would pay (asymptotically optimal)

$$H \leq L_H < H + \frac{1}{K}$$

↳ However, this would increase the size of the Huffman tree, which costs the PREAMBLE! (RESETTING SITUATION, as $T \to \infty$.)

Very large PREAMBLE!

⇒ In practice, unpractical!

NB: K is fixed, even for an infinite text!

⇒ Still, most compressors make use of Huffman in some way! (MPEG/JPEG).

[TRADE OFF between #LEAVES/symbols and #BLOCKS].

$$H \geq \frac{1}{\log_2 |\Sigma|}$$

Enough!

CANONICAL HUFFMAN CODE

Huffman code is strongly limited by 2 facts:

- It has to store the structure of the tree in the preamble (costly if alphabet Σ is large!)
- Decoding is slow, as we need to traverse the tree from root to leaf ⇒ at every edge & the path we may have a CACHE MISS.

NEW IDEA: implement a variant of the Huffman Code (CANONICAL HUFFMAN) that alleviates such problems ⇒ VERY FAST DECODING & SMALL MEMORY USAGE!

NO NEED to navigate the HUFFMAN TREE all the time!

⇒ We need to make use of EXTRA DATA STRUCTURES & operate in the following ways:

1. Compute codeword length $L(\theta)$ for each symbol $\theta \in \Sigma$ according to classical Huffman's Algorithm.
2. Construct the array num, which stores in array position num[ℓ] the #SYMBOLS having Huffman codeword of ℓ-bits.
3. Construct the array symb, which stores in the entry symb[L] the LIST of SYMBOLS having Huffman codeword of ℓ-bits.

(4) Construct the array $fc[]$, which stores in the entry $fc[l]$ the first codeword of all symbols encoded with l bits. ← TO PROCEDURE (4)

(5) Assign consecutive codewords to the symbols in $symb[l]$, starting from the codeword $fc[l]$.

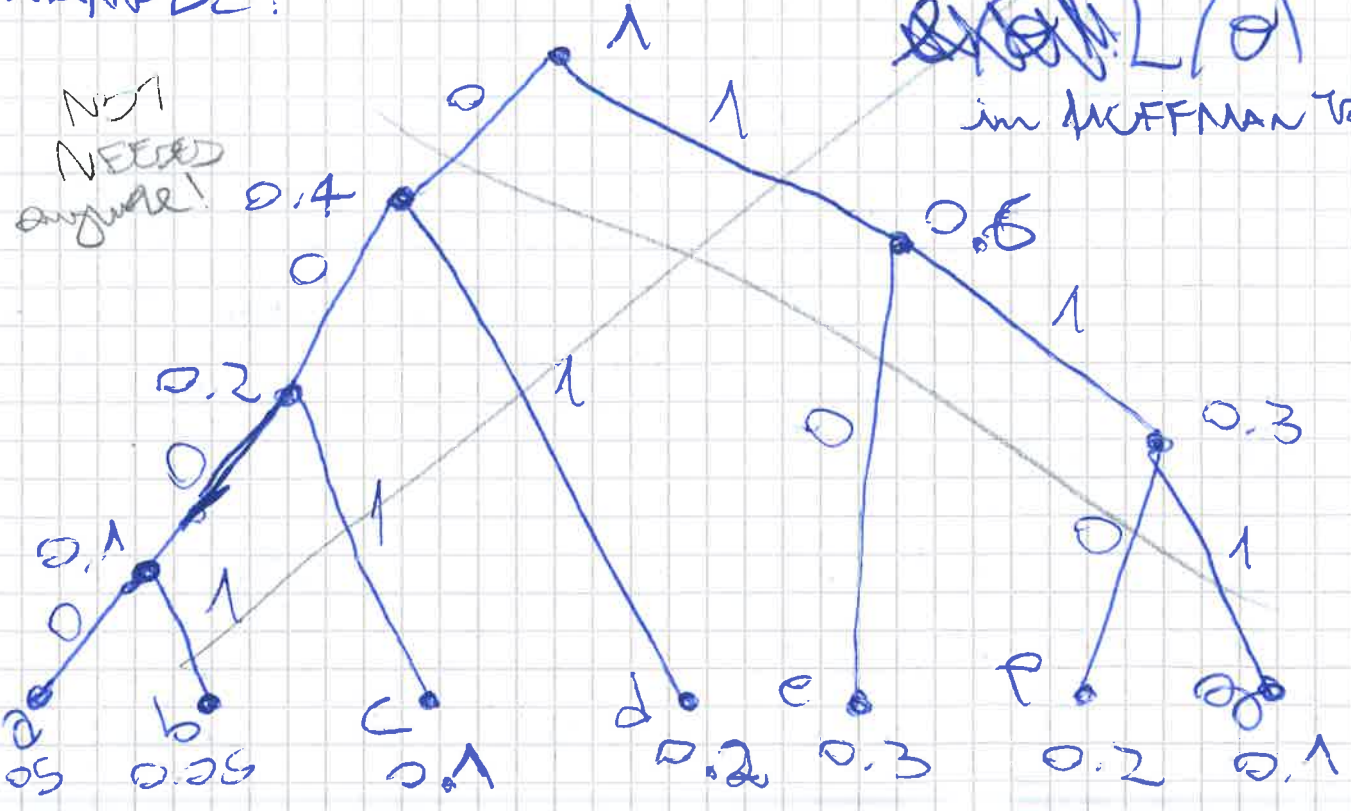
• STORAGE:

ADVANTAGE: We no longer need to store the tree-structure via pointers, but just store fc and $symb$ arrays. → Other data structures are DISCARDED

• DECODING:

We no longer need to recreate the whole Huffman tree, but we only operate on two available arrays (at most 2 CACHE MISSES per symbol instead of all labels along the path)

EXAMPLE:



$L(\alpha)$ in HUFFMAN TREE

Construct array num/ncw

1	2	3	4
0	2	3	2

$\forall$ level $\in [0, maxLevel]$

Construct array symbols

offset / level	0	1	2
1			
2	d	e	
3	c	f	g
4	a	b	

We can fit millions of symbols into cache!

Construct array fcw according to procedure (4):

1	2	3	4
2	3	1	0

fcw[l] = the value of the codeword starting at level l

$$fcw[i-1] = \frac{fcw[i] + num[i]}{2}$$

$\forall i$

? 10 001 000

DECOMPRESS

$\Rightarrow$ We can now ~~[illegible]~~ via:

```
v = next_bit(); l = 1
while( v < fcw[l] ):
    v = 2v + next_bit();
    l++;
return symbols[ l, v - fcw[l] ]
```

IDEA: When FEW reaches the right of a codeword to decode, then we actually decode. [Reaching the right level!] within the while, but only in the return.

NB: No cache misses occur

⇒ Improves ENCODING & DECOMPRESSION SPEED hugely.

If $P[x] = \frac{1}{2+x^2}$, is it still worth using HAMMING CODING? No, just apply a δ-CODE, as it is much easier to apply.

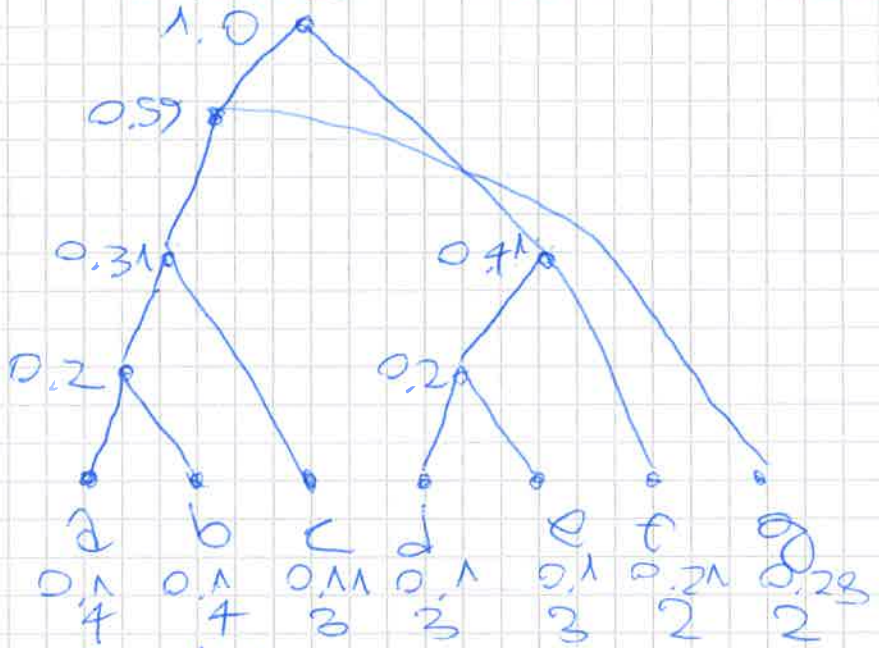
To build FCW:

```
fcw[max]=0
for(i=max-1; i>=1; i--)
    fcw[i] = (fcw[i+1] + ncw[i+1]) / 2
```

SYMBOL TABLE:

level	0	1	2	3
1		D	C	A
2		E		
3		F		
4				

CANONICAL HUFFMAN CODE - COMPRESSIONE



Build the level:

1	2	3	4
0	2	3	2

SYMBOL TABLE

level	0	1	2
1			
2	f	g	
3	c	d	e
4	a	b	

Build the δ-v:

1	2	3	4
1	2	1	0

$$CW(0) = FCW[l[0]] + OFFSET(0) \text{ in } l \text{ BITS}$$

$CW(g) = 2 + 1 = 3$ in 2 bits $= (11)_2$

$CW(f) = 2 + 0 = 2$ in 2 bits $= (10)_2 \Rightarrow FCW[2] = 2$

$CW(e) = 1 + 2 = 3$ in 3 bits $= (011)_2$

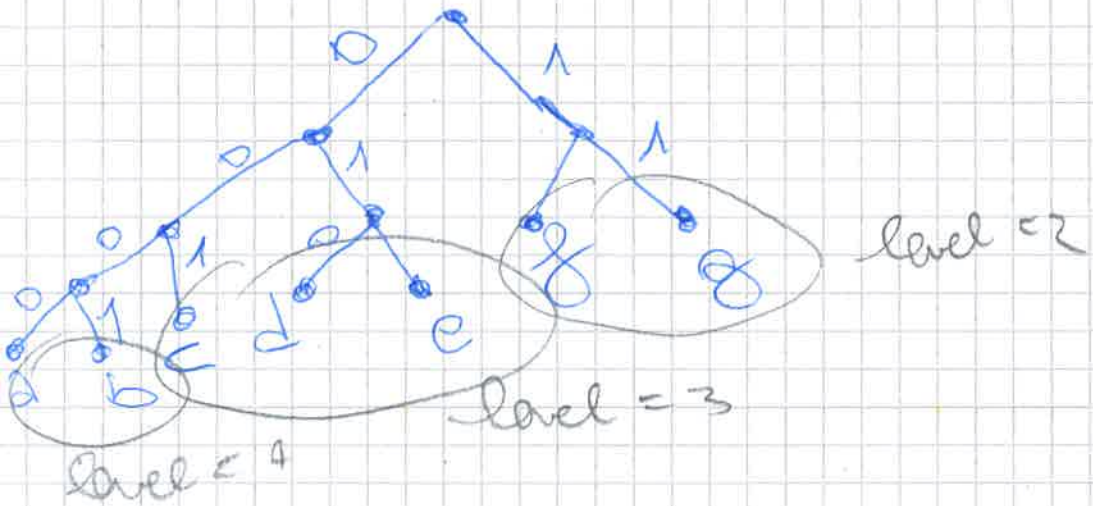
$CW(d) = 1 + 1 = 2$ in 3 bits $= (010)_2$

$CW(c) = 1 + 0 = 1$ in 3 bits $= (001)_2 \Rightarrow FCW[3] = 1$

$CW(b) = 0 + 1 = 0$ in 4 bits $= (0001)_2$

$CW(a) = 0 + 0 = 0$ in 4 bits $= (0000)_2 \Rightarrow FCW[4] = 0$

=> Alternatively, we could build a CANONICAL HUFFMAN TREE based on code lengths:



BLOOM FILTER - CLASS EXERCISE:

$k_{opt} = \frac{m}{n} \ln 2 = \lceil 16 \ln 2 \rceil$

$\varepsilon_{OPT} \approx (0.6185)^{\frac{m}{n}}$

Error

⇒ Compute size of BLOOM FILTER (m) to guarantee error of 2^{-20}

CANONICAL HUFFMAN - DECODING

```
V = next_bit();
l = 1;
while( V < FC[l] )
    V = 2V + next_bit();
    l = l + 1;
return symb[ l, V - FC[l] ]
```

We have: $S = 01$

	1	2	3	4
FC =	2	1	1	0

len

SYMBOL TABLE =

l	1	2	3
1	D	C	A
2	E		B
3	F		

1) $l = 1$ $\quad S = 01$

$V = 0$

$V < FCW[1]$

$0 < 2 \Rightarrow$ Go into the while loop

$V = 0 + 1$

$i = 2$

2) $l = 2$

$V = 1$

$V < FCW[2]$

$1 < 1$ ✗

$\Rightarrow$ Return symb$[l, V - \delta < [l]]$;

symb$[2;\ 1 - \delta CW[2]]$

symb$[2;\ 0] \Rightarrow$ (first symbol $\delta > 2$

~~IDEA: Keep scanning bits until we find~~

~~$\delta CW[l] < [l] V$~~

~~Then, we Return~~

~~SYMBOL. SYMB$[l, [i] - \delta CW[l]]$~~

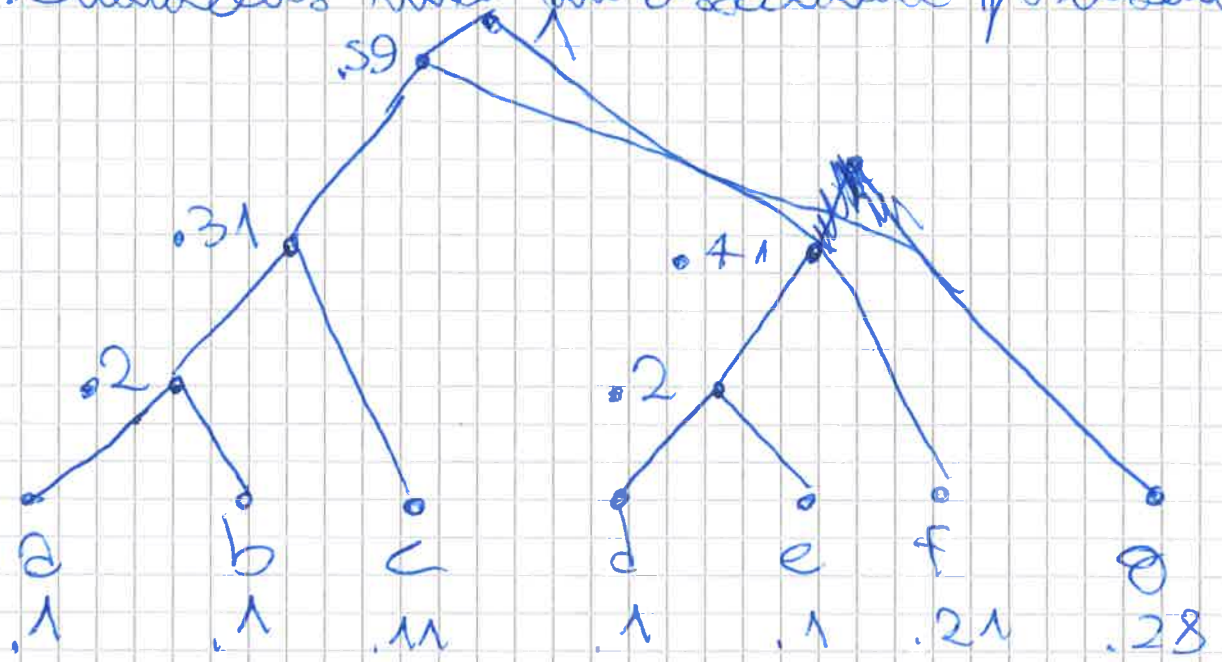
IDEA: Keep scanning

IDEA: Stop the while loop when the symbol is no longer strictly

$\rightarrow$ RETURN SYMB$[l, V - FCW[l]]$

CANONICAL HUFFMAN ~~COMPRESSION~~ & DECOMPRESSION:

1) COMPRESSION: We are given the following characters with an associated probability.



---

(0) V = 0 | V = 1 | V = 3 ⇒ return SYMB[l, V - first[l]]
l = 1 | l = 1 | l = 2 SYMB[2, 2]

(1) V = 1 ⇒ | V = 2 ⇒ return SYMB[2, 1] = g
l = 1 | l = 2

And the SYMBOL TABLE (which symbols are at which level):

symbol / level	1	2	3
1			
2	8	∞	
3	c	d	e
4	a	b	

⇒ Before decompressing, we "discard" the HUFFMAN TREE and the HCW.

HCW:

2	1	2	3
2	2	1	0

l = 0 0 0 1 1 0 1 0 1 0 1 1 1 0 1 1 1 1 0 0

① V = 0 ⇒ | = 1 ⇒ V = 0 + 0 | = 2 ⇒ V = 1 & | = 3 ⇒ LEAVE WHILE

return: SYMB[3, 1 − FC[3]]

SYMB[3, 1] = c

② V = 1 ⇒ | = 1 ⇒ V = 2 | = 2 ⇒ LEAVE WHILE

return SYMB[2, 2 − FC[2]]

SYMB[2, 1] = 8

③ V = 1 ⇒ | = 1 ⇒ V = 2 | = 2 ⇒ LEAVE WHILE

return 8

④ return 8

⑤ V = 1 ⇒ | = 1 ⇒ V = 3 | = 2 ⇒ LEAVE WHILE

return SYMB[l, V − FC[l]]

SYMB[2, 2] = ∞

ARITHMETIC ENCODING

Part of the bigger family of STATISTICAL ENCODING TECHNIQUES.

⇒ Recall that HUFFMAN Encoding was bounded by the following entropy:

$$H \leq L_H + 1 < H + 1$$

So, Huffman was not OPTIMAL for coding sources with a very small entropy (a very skewed distribution), or it uses at least one bit per symbol.

⇒ Huffman substitutes each input symbol with a codeword ⇒ AVG. length of a TEXT T compressed by Huffman is $\Omega(|T|)$

⇓

Huffman can't be better than $\frac{1}{\log_2 |\Sigma|}$ ($|\Sigma|$ = alphabet & T).

The BEST CASE is given by a substitution of each character by 1 bit.

ARITHMETIC ENCODING

No longer a PREFIX-CODE!

⇓

Compressed output is NOT a concatenation of CODE words associated to symbols of Σ, but:

"A bit of the encoded output can represent more than one input symbol".

⇓

We obtain the following:

- Better COMPRESSION, at the cost of slowing down the algorithm & lost capability to DECODE characters at any position.
  [+ Also works in a DYNAMIC MODEL, where probabilities $P\{\theta\}$ are updated as the input sequence S is processed]

HUFFMAN CODING: $H \leq L_H < H + 1$

ARITHMETIC CODING: $H \leq L_A < H + \frac{1}{n}$

BINARY REPRESENTATION AS DYADIC FRACTIONS

INPUT: A bit stream $b_1 b_2 \dots b_k$, with $k \to \infty$

can be interpreted as a real number in the range $[0,1)$ by prepending a "0." to it.

$$b_1 b_2 \dots b_k \to 0.b_1 b_2 b_3 \dots b_k = \sum_{i=1}^{k} b_i \cdot 2^{-i}$$

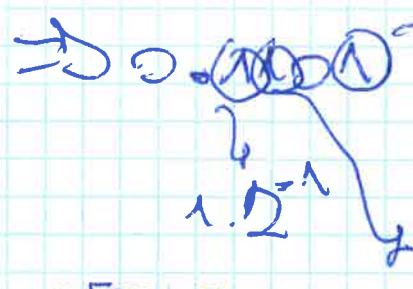
$b_1 \cdot 2^{-1}$ $b_2 2^{-2}$ $b_3 2^{-3}$ $b_4 2^{-k}$

$$\sum_{i=1}^{k} b_i \cdot 2^{-i} = \frac{V}{2^k} = \text{DYADIC FRACTION!}$$

~~We want to code X, where: $0 \leq X < 1$~~

~~EXAMPLE:~~

EXAMPLE: INPUT: 1101



$\Rightarrow 0.1101 = \frac{1}{2} + \frac{1}{4} + \frac{1}{16} = \frac{8+4+1}{16} = \frac{13}{16} = \frac{V}{2^K}$,

where K = 4 (the last decimal bit)

CONVERSION ALGORITHM:

REAL NUMBER (FRACTION) $\Rightarrow$ (possibly) (POTENTIALLY) INFINITE # BITS

(Uses to convert result CANONICAL (HUFFMAN) into BINARY)

can be converted into

According to the following algorithm:

Converter(x): // $x \in \mathbb{R}$ in $[0, 1)$

repeat
  $x = 2 \cdot x$
  if $x < 1$ then:
    output = output :: 0
  else:
    output = output :: 1 (CONCATENATION)
    $x = x - 1$
  end if
until accuracy desired

// the loop going till the desired level of accuracy is reached.

DYADIC FRACTION:

A fraction of the form $\frac{V}{2^K}$, where V and K are positive integers.

$\Rightarrow$ Real number associated to $b_1 b_2 b_3 \ldots b_K$ is the DYADIC FRACTION $\frac{val(b_1 b_2 \ldots b_K)}{2^K}$

And a FRACTION $\frac{V}{2^K}$ can be written as: $bin_K(V)$

BINARY REPRESENTATION of integer V on a bit string of length K.

EXAMPLE: $X = \frac{1}{3}$

⇒ Apply the Converter algorithm to it.

1) $\frac{1}{3} \cdot 2 = \frac{2}{3} < 1 \Rightarrow$ output = 0

2) $\frac{2}{3} \cdot 2 = \frac{4}{3} \geq 1 \Rightarrow$ output = 01

$x = \frac{4}{3} - 1 = \frac{1}{3}$

The procedure could continue forever!

$0.01010101 = 0.\overline{01}$

0.33333...

COMPRESSION by ARITHMETIC CODING.

Arithmetic coding operates in steps, one per each letter of the input sequence compressed so far, and the PROBABILITIES and the CUMULATIVE PROBABILITIES of alphabet symbols ⇒ Interval is further subdivided into smaller subintervals (one for each symbol of the alphabet)

TRUNCATION ERROR:

As we do not have an infinite # BITS at our disposal, we generally truncate somewhere our BINARY STRING.

↳ Suppose we truncate after the d+1-BIT. What is the MAX. ERROR we would get?

WORST-CASE ERROR:

All digits following the d+1-digit are 1.

$$0.1010111111111\ldots$$

TRUNCATION | ERROR

$$x = .\underbrace{b_1 b_2 \ldots b_d}_{d \text{ DIGITS}} | b_{d+1}$$

TRUNCATION

$\Rightarrow$ Truncating at b_d, we would get that x can differ from x on the bits following the truncation at most by

$$x \in [x - 2^{-d}, x]$$

$\wedge$ → TRUNCATION of x

$$\hat{x} = .b_1, b_2, \ldots, b_d \underbrace{b_{d+1} b_{d+2} \ldots b_{d+1}}_{0000\ldots000}$$

$$x - \hat{x} = .\underbrace{000\ldots0}_{b_d} \underbrace{b_{d+1} b_{d+2} \ldots b_{d+1}}_{000\ldots01111\ldots11}$$

$$\leq$$

$$.b_1 b_2 \ldots b_d \underbrace{b_{d+1} b_{d+2} \ldots b_{d+1}}_{1111111\ldots}$$

$$\#1s = \sum_{i=1}^{+\infty} 1 \cdot 2^{-(d+1)} = \sum_{i=1}^{+\infty} 2^{-d} \cdot 2^{-i}$$

(following) WORST-CASE

$$= 2^{-d} \cdot \sum_{i=1}^{+\infty} 2^{-i} = 2^{-d} \cdot \sum_{i=1}^{+\infty} \frac{1}{2^i} = 2^{-d}$$

(GEOMETRIC SERIES) $= 1$

$\Rightarrow$ The error reduces exponentially with the # bits used.

COMPRESSION (CODING) BY ARITHMETIC ENCODING

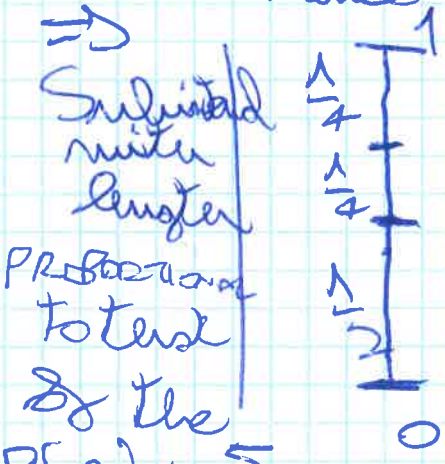
INPUT: - A text T;
- Alphabet Σ, with probabilities $P\{\sigma\}$ associated to each character σ

$T = abac$

$\Sigma = a\ b\ c$ (probabilities $\frac{1}{2}\ \frac{1}{4}\ \frac{1}{4}$)

$\Rightarrow$ We hence have:

Subdivided with respect to probabilities to test $P\{\sigma\}$ in Σ.



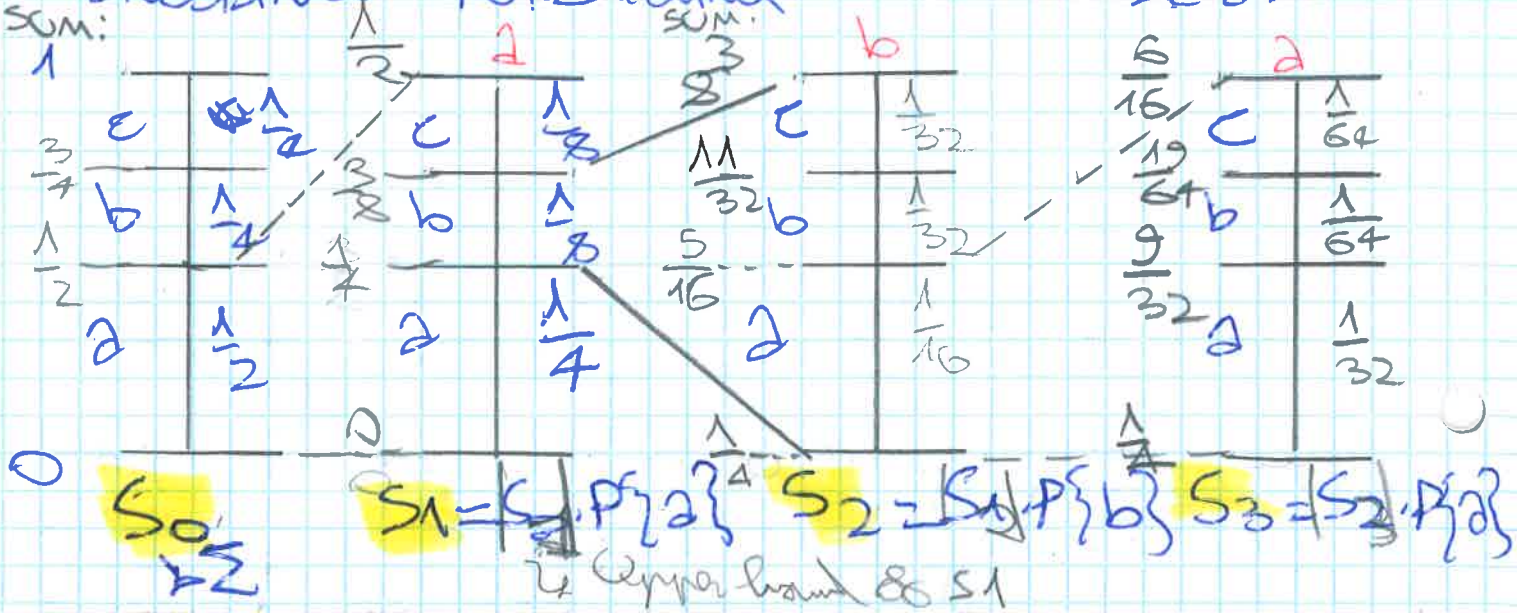
FOR Σ

$\Sigma = S_0$

$S_1\ S_2\ S_3\ S_4$

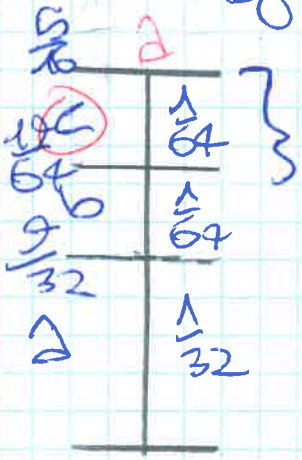
$T = a\ b\ a\ c$

ENCODING FUNCTIONING - VISUALIZED:



~~Encode character Present~~

Process each character of the text T, expanding the corresponding Θ in S_i $\Rightarrow$ the intervals are given by the sum of the P. up to that character



FINAL INTERVAL for (2)

abac

$\Rightarrow T \in \left[\frac{19}{64}, \frac{5}{16}\right]$ RESULT!

↳ This number is inside all previously-generated intervals.

↓

Starting from this number, together with (W), we can proceed backwards and re-construct the ranges.

↳ After performing 4 steps backwards we stop.

DECOMPRESSION by ARITHMETIC DECODING

INPUT: • Result of compression stage + $\left[\frac{19}{64}, \frac{5}{16}\right]$

• $|T|$

• Symbol probabilities $P\{\Theta\}$ & Σ.

OUTPUT: Original sequence T.

FUNCTIONING:

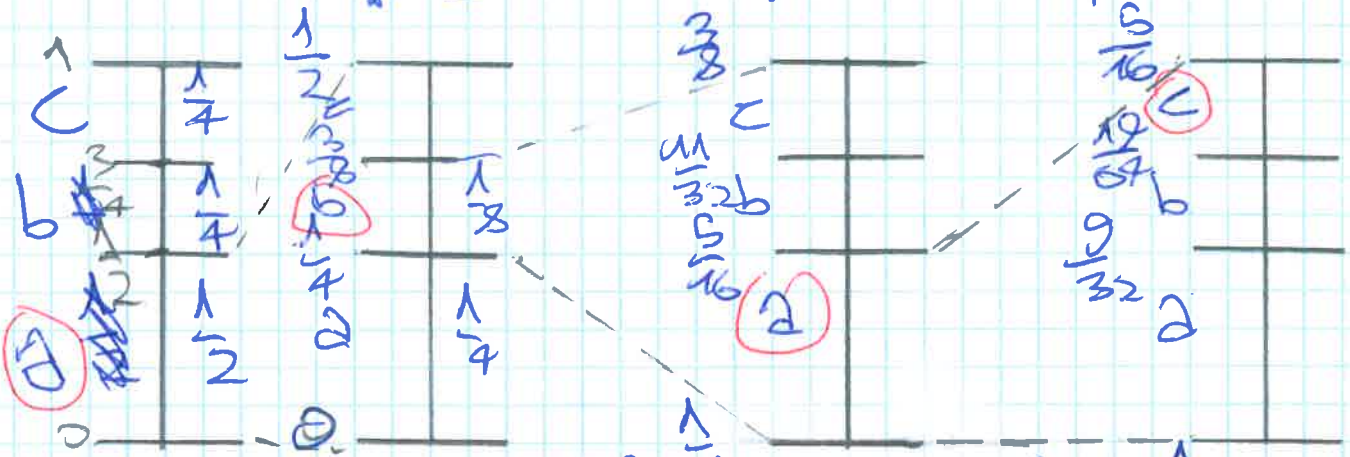
• In the first iteration, we make use of the Distribution of $P\{\Theta\}$ over Σ.

- In every iteration, we decode one symbol using the current model; we then update the model.

EXAMPLE: DECOMPRESSION.

ENCODED RESULT $= \left[\frac{39}{128}, 14\right]$ $\Sigma = \{a, b, c\}$

$P\{a\} = \frac{1}{2}$; $P\{b\} = \frac{1}{4}$; $P\{c\} = \frac{1}{4}$;



$S_0 = P\{\}$ $S_1 = S_0 \cdot P\{a\}$ $S_2 = S_1 \cdot P\{b\}$ $S_3 = S_2 \cdot P\{a\}$

$P\{\}$

Where does $\frac{39}{128}$ lie? Where does $\frac{39}{128}$ lie? Where does $\frac{39}{128}$ lie? Where does $\frac{39}{129}$ lie?

in $\left[0, \frac{1}{2}\right]$ in $\left[\frac{1}{4}, \frac{3}{4}\right]$ in $\left[\frac{1}{4}, \frac{5}{16}\right]$ in $\left[\frac{19}{64}, \frac{5}{16}\right]$

output: a output: b output: a output: c

⇒ The reconstructed sequence T is hence $T = \{a, b, a, c\}$

[After 4 stages], we are done.

$n = |T|$

DECOMPRESSION BY ARITHMETIC ENCODING - EXERCISE.

Make use of DYADIC FRACTIONS, not reals!

$P\{a\} = \frac{1}{2} \quad P\{b\} = P\{c\} = \frac{1}{4}$

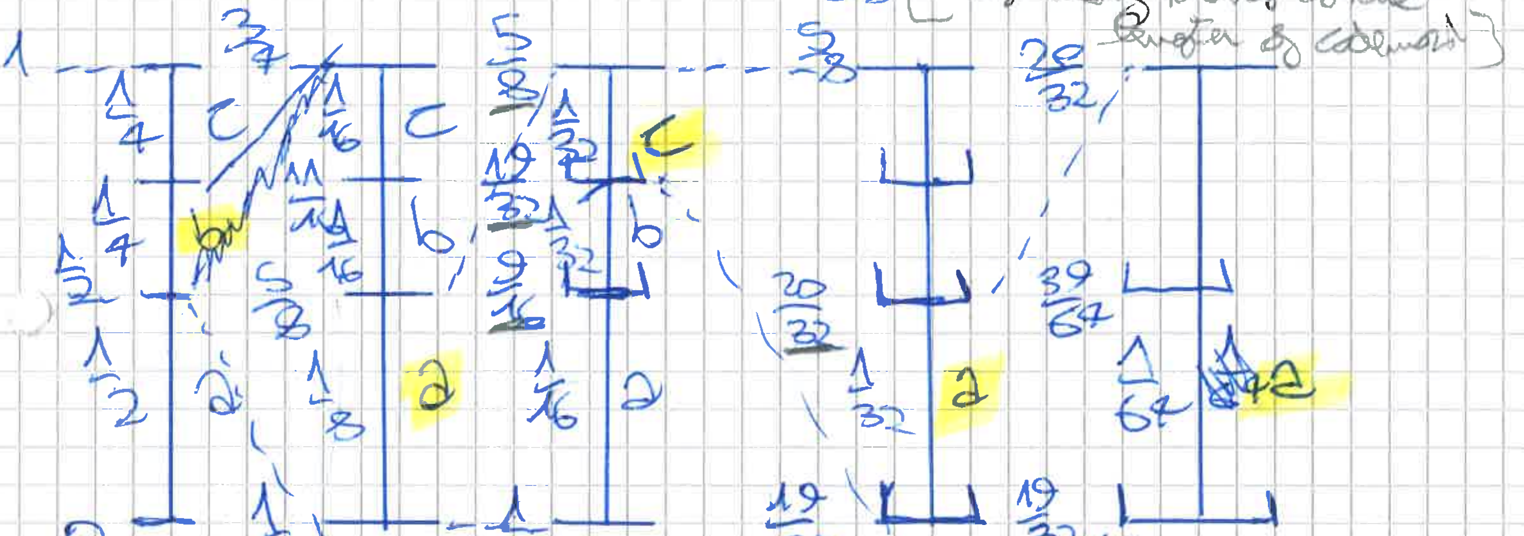
$C = 1\,0\,0\,1\,1 = \frac{16+2+1}{32} = \frac{19}{32}$

$\frac{1}{2} \quad \frac{1}{4} \quad \frac{1}{8} \quad \frac{1}{16} \quad \frac{1}{32}$

First thing ⟹ Convert into a DYADIC FRACTION

1) Then, proceed analogously to arithmetic encoding (but opposite)

2) Find out where $\frac{19}{32}$ lies every time (as many times as the length of codeword)



$\frac{1}{2} + \frac{1}{4} = \frac{3}{4}$

$\frac{1}{2} + \frac{1}{8} = \frac{4+1}{8} = \frac{5}{8}$ $\quad \frac{2}{8} + \frac{1}{4} = \frac{5+2}{8} = \frac{7}{8}$

$\frac{9}{16} + \frac{1}{32} = \frac{18+1}{32} = \frac{19}{32}$

$\frac{1}{2} + \frac{1}{16} = \frac{8+1}{16} = \frac{9}{16}$

T = b a c a a

$\frac{19}{32} + \frac{1}{64} = \frac{38+1}{64} = \frac{39}{64}$ $\quad \frac{19}{32} + \frac{1}{32}$

T = b a c a

EFFICIENCY & ARITHMETIC ENCODING

After finding $[l_n, s_n]$ for an input sequence s^*, we still need to find an efficient way to encode it.

ENCODING OF $[l_n, s_n]$

(*e: of the RANGE found through arithmetic encoding)

⇒ We would like to pick a number in the interval $[l_n, s_n]$ such that it can be encoded with as few bits as possible.

Notice that ⇒ We have shown that:

$$x - \hat{x} \leq 2^{-d} \Rightarrow x - 2^{-d} \leq \hat{x}$$

(TRUNCATION of x after d bits)

and $\hat{x} \leq x$

$\Downarrow$

Differing bits have been set to 0

$\leq s_n$

$[l_n, \quad]$

How do we encode it?

$\Downarrow$

$x = .b_1 b_2 \ldots b_d b_{d+1} \ldots b_{d+n}$

$\hat{x} = .b_1 b_2 \ldots b_d 000\ldots0$

ENCODING BY "TRUNCATION"

$[l_n, s_n] \Rightarrow [l_n, l_n + s_n]$

⇒ Truncating $l_n + s_n$ to its $\lceil \log_2 \frac{2}{s_n} \rceil$ bits gives a number in $[l, (l + s_n))$

$l = l_n ; s = s_n$

COROLLARY:

$[l, s] \Rightarrow [l, l+s]$ can be truncated. Where?

$\Rightarrow$ Truncate $l + \frac{s}{2}$ to its first $\lceil \log_2 \frac{2}{s} \rceil$

This would make it "fall" in the interval $[l, l+s]$

PROOF: Set $d = \lceil \log_2 \frac{2}{s} \rceil$

$$2^{-d} = \frac{1}{2^d} \leq \frac{1}{2^{\log_2 \frac{2}{s}}} \leq \frac{1}{\frac{2}{s}} \leq \frac{s}{2}$$

$$\Rightarrow 2^{-d} \leq \frac{s}{2}$$

RETURN VALUE: SIZE

$\Rightarrow$ ARITHMETIC ENCODING has hence enough to return the first $\lceil \log_2 \frac{2}{s_n} \rceil$ BITS (TRUNCATION)

& the BINARY REPRESENTATION of value: $l_n + \frac{s_n}{2}$ with $\lceil \log_2 \frac{2}{s_n} \rceil$ BITS.

EXAMPLE: We had found that the resulting interval was: $\left[\frac{12}{64}, \frac{5}{64}\right]$ 16

$$\Rightarrow \left[\frac{12}{64}, \frac{20}{64}\right] = [l_n, s_n]$$

$\Rightarrow \left[\underbrace{\frac{19}{64}}_{l_n} ; \underbrace{\frac{19}{64}}_{l_n} + \underbrace{\frac{1}{64}}_{s_n}\right]$ $\quad l_n = \frac{19}{64}$, $s_n = \frac{1}{64}$

$\Rightarrow$ OUTPUT of COMPRESSION STAGE

$\Rightarrow$ BINARY REPRESENTATION of $l_n + \frac{s_n}{2}$

$$l_n + \frac{s_n}{2} = \frac{19}{64} + \frac{1}{64} \cdot \frac{1}{2}$$

$$= \frac{19}{64} + \frac{1}{128} = \frac{38+1}{128} = \frac{39}{128}$$

$\Rightarrow$ We now need to represent $\frac{39}{128}$ bits making use of $\log_2 \frac{2}{s_n}$ BITS.

$$\log_2 \frac{2}{\frac{1}{64}} = \log_2 128 = 7$$

$\Rightarrow$ We will make use of 7 BITS!

How to find these 7 BITS?

$\Rightarrow$ CONVERT $\frac{39}{128}$ in BINARY (converter ALGORITHM) and truncate after 7 BIT! 7 times!

$\Rightarrow \frac{39}{128} \cdot 2 < 1 \Rightarrow$ OUTPUT $= 0$

$\frac{78}{128} \cdot 2 < 1 \Rightarrow$ OUTPUT $= 1$

$\left(\frac{156}{128} - 1\right) \cdot 2 < 1$

$\frac{50}{128} < 1 \Rightarrow$ OUTPUT $= 0$

$\frac{112}{128} \cdot 2 < 1^2 \Rightarrow$ OUTPUT $= 1$

$\left(\frac{224}{128} - 1\right) \cdot 2 = \frac{192}{128} \geq 1 \Rightarrow$ OUTPUT $= 1$

$\left(\frac{192}{128} - 1\right) \cdot 2 = 1 \geq 1 \Rightarrow$ OUTPUT $= 1$

$\Rightarrow$ FINAL OUTPUT:

$< \underbrace{0\,1\,00\,11}_{\text{PRIMARY RES. OF } T.}, 4 >$ $4|T|$

PROOF of the RELATION of #BITS emitted by ARITHMETIC ENCODING and the ENTROPY. (i.e. it is ≥ 2 BITS from the OPTIMAL LOWER BOUND)

(i.e.: Relation with #BITS)

We know that the SIZE of a string $S_n = \prod_{i=1}^{n} P\{\underbrace{T[i]}_{\text{CHAR. AT POS. } i}\}$

And we have seen that S_n will be represented by $\log_2 \frac{2}{S_n}$ BITS.

$$\lceil \log_2 \frac{2}{S_n} \rceil < \log_2 \frac{2}{S_n} + 1$$

BITS with which we encode ARITHMETIC CODING

$$= \log_2 2 - \log_2 S_n + 1$$

$$= 2 - \log_2 S_n$$

$$S_n = \prod_{i=1}^{n} P\{T[i]\}$$

$$= 2 - \log_2 \left(\prod_{i=1}^{n} P\{T[i]\} \right)$$

BY LOGARITHMS' LAW

$\log_2(a \cdot b) = \log_2 a + \log_2 b$

P. error at pos. i

$$= 2 - \sum_{i=1}^{n} \log_2 P\{T[i]\}$$

n_θ = # occurrences of θ

the group some symbol

$$= 2 - \sum_{\theta} n_\theta \cdot \log_2 P\{\theta\}$$

$P\{\theta\} = \frac{n_\theta}{n}$ is n not so big

DIV. & MULT. by n.

$$= 2 - n \cdot \sum_{\theta} \frac{n_\theta}{n} \cdot \log_2 P\{\theta\}$$

taken in!

$$= 2 - n \cdot \sum_{\theta} P\{\theta\} \cdot \log_2 P\{\theta\}^{-1}$$

We know the DEFINITION of ENTROPY:

$$H = \sum_{\theta} P\{\theta\} \cdot \log_2 \left(\frac{1}{P\{\theta\}} \right)$$

$$\Rightarrow \boxed{\lceil \log_2 \frac{2}{S_n} \rceil < 2 + n \cdot H_0}$$

ARITHMETIC ENCODING:

Now: we can get the length of every symbol by dividing by (n).

$$L_A = \frac{nH}{n} + \frac{2}{n}$$

$$L_A = H + \frac{2}{n}$$

or:

$$\Rightarrow \lceil \log_2 \frac{2}{Sn} \rceil < 2 + n \cdot H$$

$\Rightarrow$ There is a waste of just 2 BITS for an input sequence S, or $\frac{2}{n}$ per symbol of the sequence.

These 2 BITS are hence negligible for longer and longer sequences.

$\Rightarrow$ HUFFMAN:

$n + n \cdot H$

$\Downarrow$

CANONICAL: Is fast to create & can decompress any partition.

$\Downarrow$

Uses whole bit symbols

ARITHMETIC.

$2 + n \cdot H$

$\Downarrow$

Better than H!

$\Downarrow$

+ On the fly!

$\Downarrow$

Only entire file is decompressable.

$\Downarrow$

Generally used when highly skewed DSR.

13 DICTIONARY-BASED COMPRESSORS

Different approach from the statistical one! ⇒ Not interested in PROBABILITIES of source S!

D = "cane" → "c"
"gatto" → "g"
"mento" → "z1"

Dictionary & MAPPING (each string replaced via a token)

S = "cane", "mento", "gatto"
S' = "c", "z1", "g"

The DICTIONARY is of GREAT IMPORTANCE in achieving good compression!

(Ex: Italian text dictionary vs English text)

ZIV & LEMPEL (LZ − xx) introduced a series of compressors, where each (already seen) string occurrence is substituted with:

- Either the offset of the previous occurrence
- ID assigned incrementally to new dictionary phrases ⇒ DYNAMIC DICTIONARY.

Starts empty and (as input sequence is processed)
(beginning: low compression, but after some Kbs good compression is obtained.
≈ 20% compression ratio.

LZ77:

Performance comparable to that of ARITH-METIC & HUFFMAN coding (according to the authors).

ALGORITHMIC GROUNDBASE:

Sliding Window $W[1,w]$, containing a portion of the input sequence that has been processed so far [typically last w characters] and look-ahead buffer B, which contains suffix of the text still to be processed.

Ex:

$W = aabbabaabb$

$B = baababaabbaa\$$

aabb abab baababaabbaa$

W B

ALGORITHM proceeds in 2 PHASES:

- PARSING: Transform the INPUT sequence into a sequence of triples of integers.
- ENCODING: Turns triples into a (compressed) bit-stream by applying a STATISTICAL & COMPRESSED encoding scheme.

FUNCTIONING VISUALIZED:

W | aabbabab ⟹ <0, 0, a>
distance of copy — length
aabbabab ⟹ <1, 1, b>
aab|babab ⟹ <1, 1, a>
aabba|bab ⟹ <2, 3, EOF> → END of STRING REACHED

PARSING STAGE'S FUNCTIONING:

Search for the longest prefix α of B that occurs as a SUBSTRING of WB (CANNOT just be in B) (not B, as the previous occurrence (CONCATENATION, interleave!) we are searching for may start in W and extend up to B).

⟹ If α occurs at distance d within from cur. position (beginning of B) and it is followed by c, then we emit:

<d, |α|, c>

distance of α from the beginning of B · size of α · char following α

If: <0, 0, B[1]> if no match!

φ, W | aabbabab ⟹ <0, 0, a>
a|abbabab ⟹ <1, 1, b>

⟹ If a match of α of B in WB is found, we advance the window by |α| + 1 positions in B and slides W correspondingly.

⟹ If a match is NOT found, the output triple is <0, 0, B[1]>, ...

SLIDING WINDOW: Delimits the size of the dictionary (quadratic in W's length)

W = ∅ aabbabab ⇒ <0, 0, a> (d, |α|, c; B)

W = a | a abbabab ⇒ <1, 1, b>

aab | babab ⇒ <1, 1, a>

aabba | bab ⇒ <2, 3, EOF>

↳ Performance of LZ77 is strongly dependent on W's length [WINDOW]

↳ The shorter W, the worse the compression ratio [faster compression time]

↳ The larger W, the better the compression ratio [but the longer the compression time]

LZSS (1982): In this version, we improve over LZ77, by avoiding to add the character c following α. If it's not found, we not emit this 0s all the time.

⇒ SOLUTION: Just emit:

- If α found in W B: <d, |α|> ⇒ Advance W by |α|
- If α not found in W B: <0, B[1]>

EXAMPLE:

B — distance of d from the beginning of B.

|aabbabab ⇒ <0, a>

a|abbababa ⇒ <1, 1>

aa|bbababa ⇒ <0, b>

aab|bababa ⇒ <1, 1>

aabb|ababa ⇒ <3, 2>

aabbab|ab ⇒ <2, 2>

⇒ We can then ENCODE the emitted pairs by STATISTICAL ENCODING or by INTEGER ENCODING.

[GZIP is based on LZ55]

Hash tables

LZ78:

Q: DISTANCE computed as?

Sliding window of LZ77 speeds up search for longest phrase to encode, but also limits the search space (and hence the ultimate compression ratio)

⇒ Build INCREMENTALLY an EXPLICIT DICTIONARY that contains a subset of strings of the input sequence S,

D = Dictionary where every phrase f is identified via integer ID(f).

⇒ We then parse S, and determine the longest prefix f' that is also a phrase of D.

=> We then substitute the sub-string phrase by pair <ID(f'), c>, where c is the character following f' in S'.

=> Steps 2 & 3 will then be enabled by STATISTICAL OR VARIABLE LENGTH INTEGER encoding.

EXAMPLE D:

PHRASE INPUT	ID.
abcd	43
abcde	44

S = abcdefg

(1) Output: <43, e>

=> Adds abcde, e -> 44 to D.

EXAMPLE 2:

=> We construct a TRIE based on the BIGRAMS output.

EXAMPLE 2. S' = Sequence yet to be parsed.

D = Current dictionary, where every phrase β is identified via integer $id(\beta)$

PARSING OF S' (part yet to be parsed). Consists of determining the longest prefix β' that is also a phrase of D, and substituting it with the pair:

LEAF: $\langle id(\beta'), c \rangle$, where c is the character following β' in S'.

Looks up in the dictionary

PHRASE: $\beta' \cdot c$

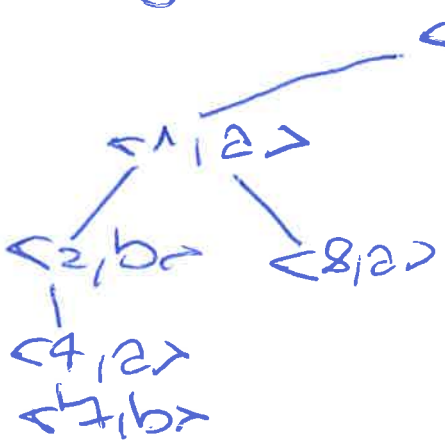
$S = a|ab|b|aba|bb|bab|aa$

INPUT	OUTPUT	DICTIONARY	LEAF TO ADD:
ε	—	0: ε	$\langle 0, \varepsilon \rangle$ ✓
a	$\langle 0, a \rangle$	1: a	$\langle 1, a \rangle$
ab	$\langle 1, b \rangle$	2: ab	$\langle 2, b \rangle$
b	$\langle 0, b \rangle$	3: b	$\langle 3, b \rangle$
aba	$\langle 2, a \rangle$	4: aba	$\langle 4, a \rangle$
bb	$\langle 3, b \rangle$	5: bb	$\langle 5, b \rangle$
ba	$\langle 3, a \rangle$	6: ba	$\langle 6, a \rangle$
abab	$\langle 4, b \rangle$	7: abab	$\langle 7, b \rangle$
aa	$\langle 1, a \rangle$	8: aa	$\langle 8, a \rangle$

$\langle ID, c \rangle$

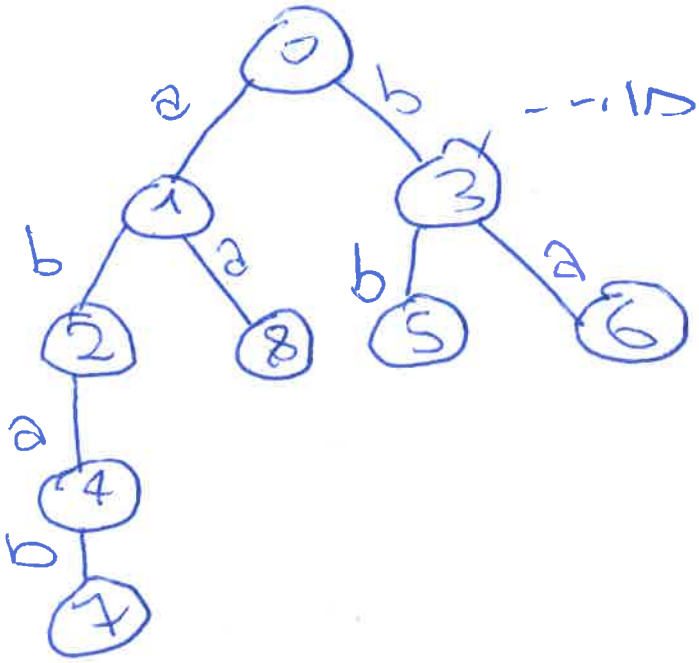
PHRASE: String of symbols in the dictionary D.

⇒ Based on these tables, we can now build the corresponding TRIE:



NB: The content the TRIE based on the occurrence of the ⓒ in the DICT.

This tree can also be represented in an UNCOMPACTED FASHION:



GZIP - SMART & FAST IMPLEMENTATION OF LZ77 (HASH-TABLE-based)

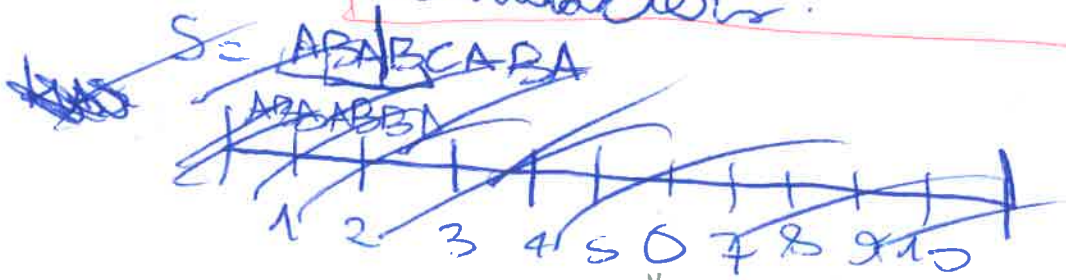
KEY PROBLEM: Fast search aimed at for longest prefix d of B that repeats in W.

↳ "BRUTE-FORCE APPROACH": Check the occurrence of every prefix of B in W via a linear BACKWARD SCAN, every single time.

↳ HASH-TABLE-based APPROACH.

We make use of a HASH TABLE to determine d and find its previous occurrence in W.

↳ KEY IDEA: Store all 3-grams occurring in W (the sliding window), namely all triplets of contiguous characters.



EXAMPLE:

3-GRAM: $S = abababcabab\ldots$

HASH TABLE:

↳ The 3-GRAMS act as "ANCHORS" to be searched for in the table.

⇒ Hash table saves all the multiple positions of the 3-gram at its corresponding key (sorted by increasing position in S)

⇒ When we shift to the right once, we output <d, l>, and we delete the 3-grams starting at V[1, l], and insert the new 3-grams starting at B[1, l]

[i.e: delete the 3-grams and so for, up to (l), and insert new 3-grams after l]

Why 3-grams?

- If 2-(B) -gram: We would have to consider too many positives
- 4-gram: We would have lots of overlappings.

SEARCH for a 3-gram a in V.

- First, search B[1, 3] in the hash table.
  - If B[1, 3] does not occur in the hash table, then (ZIP) emits <0, B[1]> and parsing advances by one char.
  - If B[1, 3] does occur in the hash table, then the list L of occurrences of B[1, 3] in V is determined.
- Once an L is found (i.e. array of positions of B[1, 3] in S), for each position (i) in L the algo. compares character-by-character

S[i,n] compared with B to compute the LCP (Longest Common Prefix).

∃ Found $(\lambda^*) \in L$ (i.e.: list of characters sharing Longest Common Prefix)

$i^* = INDEX$ of α

- p = current position in B in S.

⇒ Algorithm emits pair:

$\langle p - i^*, |\alpha| \rangle$

[GOAL: Find the LONGEST (copy)]

OPTIMAL SOLUTION: Based on tree = SUFFIX TREE of the entire text (actually not used in Practice).

⇒ Balance between theory & Practice.

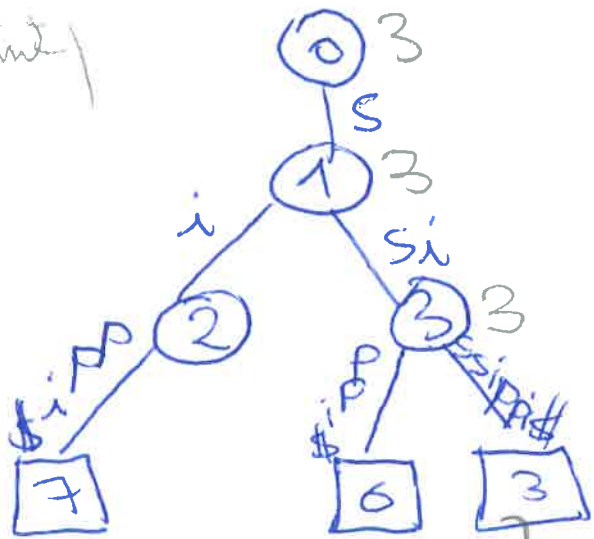
ENGINEERING COMPROMISES

[Trade-off between optimality & use-constraints]

SUFFIX TREE - based APPROACH:

(1) Build suffix tree over T. T = mississippi$

(only relevant part)



LONGEST SUFFIX (Leaf repeats)

(2) Pre-compute the MINIMUM in every single leaf among a node's leaves children.

(3) When searching for a value in the SUFFIX TREE, we keep going down as long as

minimum(node) < i. [i.e: find deepest node (i)]

TIME: O(n) => Go down as much as string's length

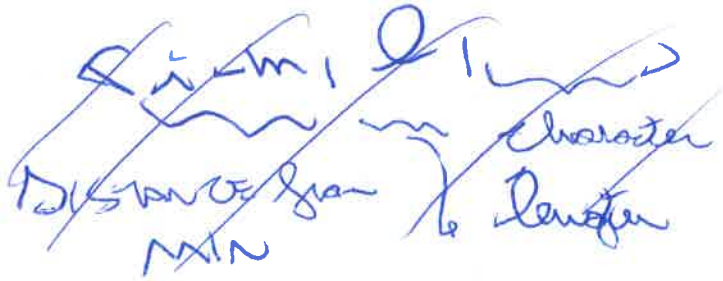
SPACE: O(n)

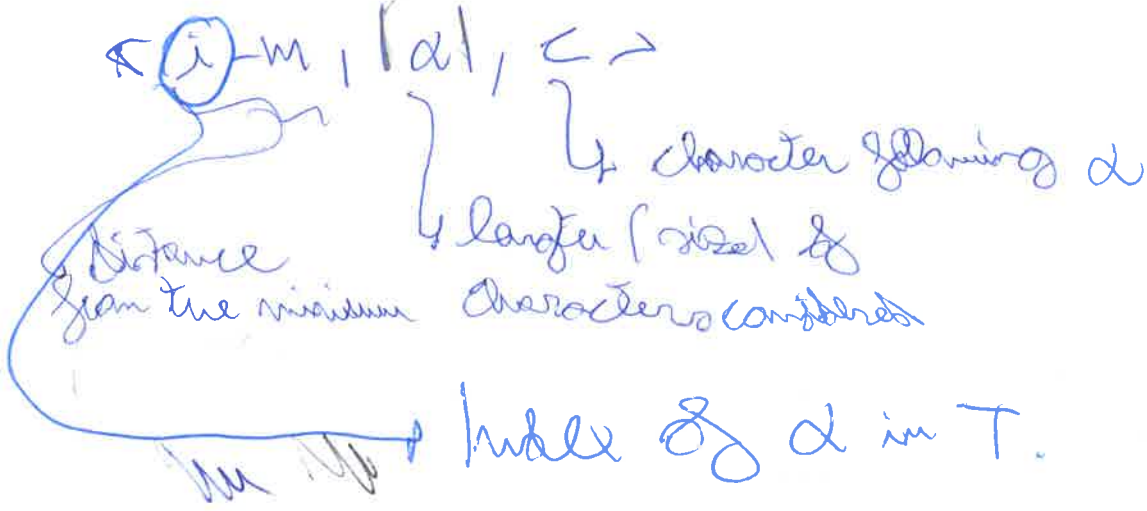
=> Need to save all the strings of T.

⚠ #COMPARISONS is proportional to a string's length.

i = current POSITION.

We emit:

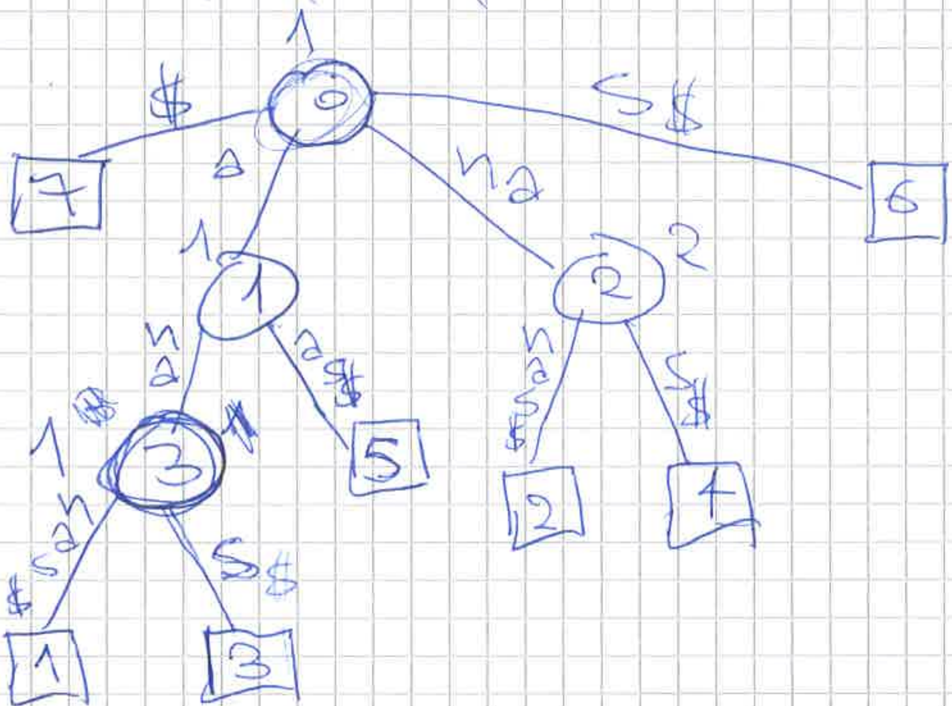




LZ77 - PARSING based on SUFFIX TREES:

(1) BUILD SUFFIX TREE based on the input string + $

T = ananas $
1 2 3 4 5 6 7



(2) We are given a DATA STRUCTURE that allows us to compute the MIN of each node among its leaves and its descendants' leaves.

(3) We can now compute the LCA for each character (go down as long as

a α = ∅	n α = ∅	a α = ana
⇓ 1	⇓	d(n) ⇓ min(n)
<0, 0, a>	<0, 0, n>	<3-1, 3, s>
		<2, 3, s>

distance of α from beginning

<DISTANCE, |α|, c>

LZ77 PARSING

↳ i - m

NB: Same result as "standard"

⇒ Go down as long as MIN(n) < i characters match with α

NORMAL LZ77 PARSING:

T = ananas $

(1) T = | ananas (β = ananas)

$\mathcal{V} = \phi \Rightarrow$ Longest prefix α of β that is a substring of $\mathcal{V}\beta$.

$\Rightarrow \langle 0, 0, a \rangle$ " ϕ

(2) T = a|nanas

$\mathcal{V} = a$

$\alpha = \phi$

$\Rightarrow \langle 0, 0, n \rangle$

(3) T = an|anas ($\mathcal{V}$ = an, β = anas)

$\alpha = ana$

$\Rightarrow \langle 2, 3, s \rangle$

We keep going down as long as:

$m \leq i$ (and chars match along the way), where:

- i = starting pos. of α in T.
- $m = \min(v) \Rightarrow$ value of the minimum at position v.

RETURN: $\langle i - m, |\alpha|, c \rangle$

TREAPS:

A TREAP is a BINARY SEARCH TREE with a modified way of ordering the nodes. Each node has a KEY (BST PROPERTIES) and a PRIORITY (MIN-HEAP PROPERTIES).

Note:

ASSUMPTION: All priorities & all keys are DISTINCT as from one another.

We hence have treat nodes respect the following properties:

- If V is a left child of U, then:

$$key(V) < key(U)$$

BST PROPERTY

- If V is a right child of U, then:

$$key(V) > key(U)$$

BST PROPERTY

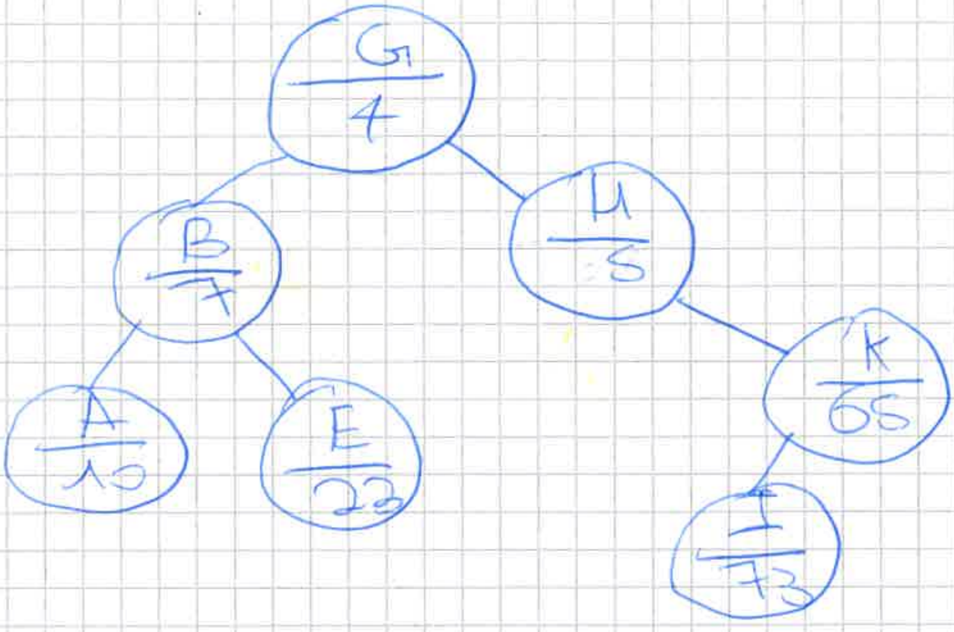
- If V is a child of U, then:

$$priority(V) > priority(U)$$

MIN-HEAP PROPERTY

HINT for TREAP: If we insert nodes $X_1, X_2, \ldots, X_n$ with associated keys into a TREAP, then the resulting TREAP is the tree that would have been formed if nodes had been inserted into a normal BST in the order giving their priority[X_i] < priority[X_{i+1}] → Node X_i inserted before X_{i+1}

EXAMPLE TREAP:



OPERATIONS ON TREAPS

- INSERT(K) — O(depth) — Used for building the TREAP, one element at a time.

↳ FUNCTIONING: STANDARD BST INSERTION

1. ↳ Follow the ~~BST~~ to insert node according to the key (as a LEAF) ✓
2. ↳ Fix the min-heap property, ensuring the PRIORITY property is respected.

→ O(1)

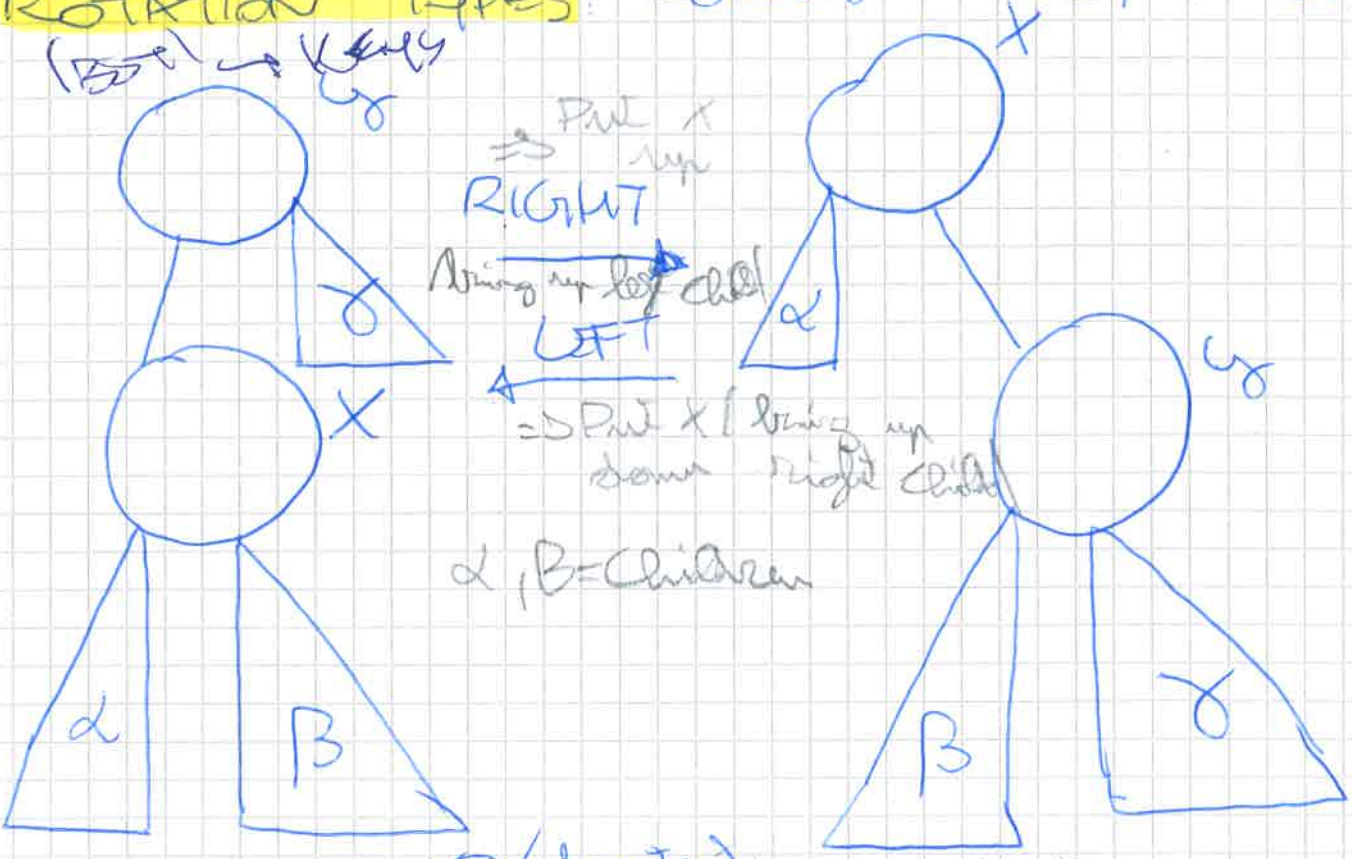
↳ ROTATION: As long as priority(c) ~~#~~ < priority(p) we perform a ROTATION, which

Get the parent down ← - ~~[crossed out]~~ increases parent (p)'s depth by 1

Get the child up ← - ~~[crossed out]~~ Decreases child (c)'s depth by 1.

And yet, we still maintain BST's ~~[crossed out]~~ properties while doing this.

ROTATION TYPES: Rotations take O(1) each.



- SEARCH(k): → O(depth) → # nodes to search
Need to navigate whole tree up to its MAX. depth (*the Treap is NOT necessarily balanced.

⇒ Operates like a SEARCH over a BINARY SEARCH TREE:

- [OPTIONAL] priority ≤ Θ.

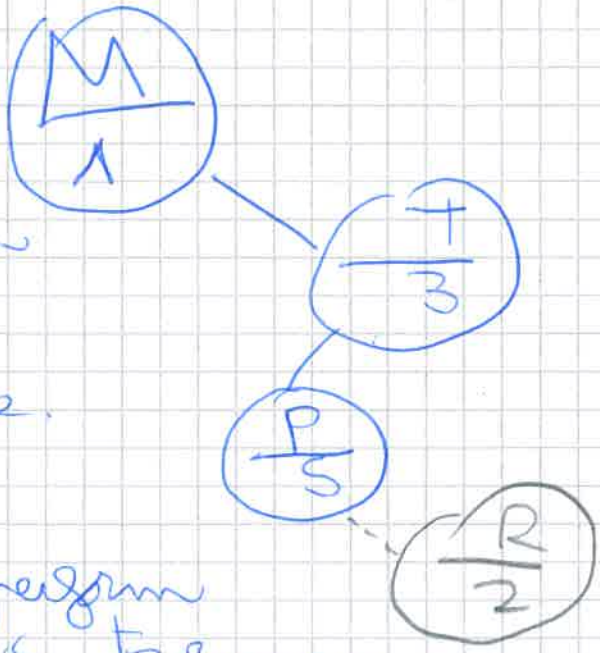
Bound the recursion (i.e. go down as long as the condition is satisfied).

NB: We cannot search by a priority key though!

Because we are dealing with a MIN-HEAP data structure.

EXAMPLE: INSERTION:

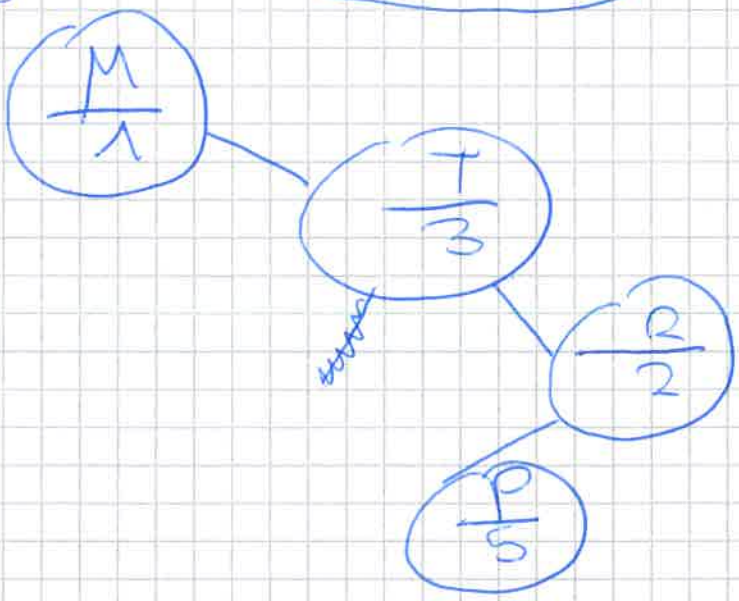
We want to insert $\frac{R}{2}$ = 4 into:



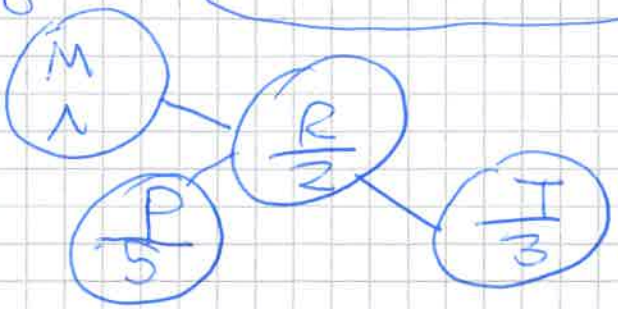
1) Search for the right LEAF POSITION in the TREE & add it there.

2) Need to perform ROTATIONS to ensure B/N-Search structure is preserved, as well as MIN-HEAP property

⇒ perform a ~~ROTATION~~ LEFT ROTATION.



⇒ perform a RIGHT ROTATION



→ O(depth)

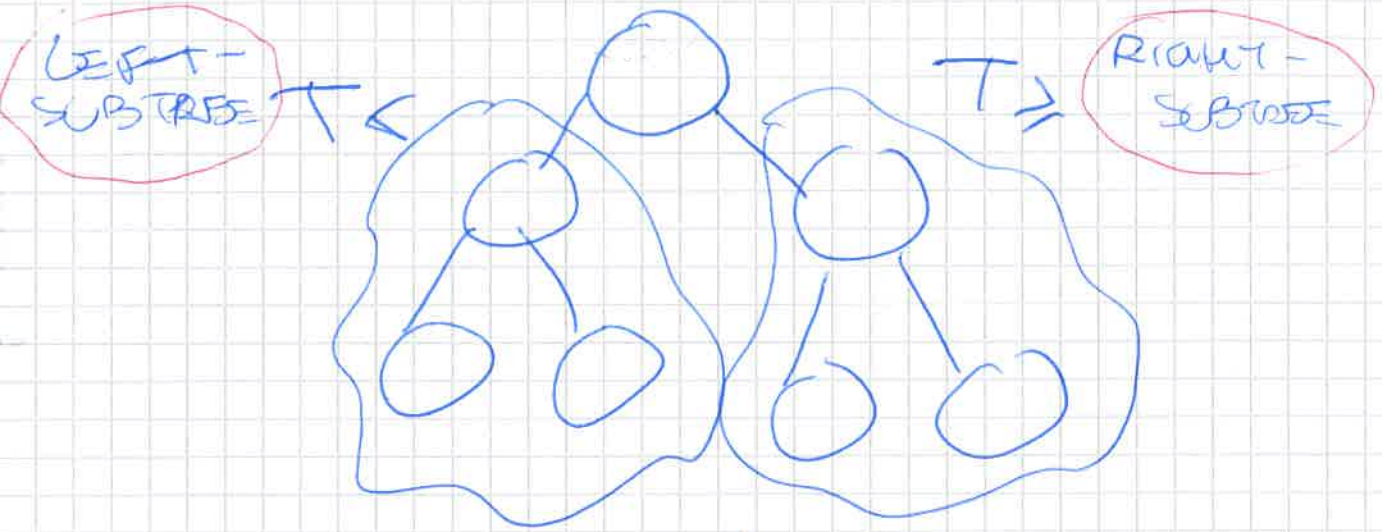
- DELETE(K): Opposite of INSERT(K).
  ⇒ Push the node down to make it a leaf via ROTATIONS ⇒ Then remove it safely.

  Ex: DELETE $\left(\frac{B}{2}\right) \Rightarrow \left(\frac{R}{\infty}\right)$ ⇒ Set No PRIORITY to $+\infty$

  Will need to be pushed down as a LEAF, then can be safely removed.

→ O(depth)

- SPLIT(K) ⇒ If K == root ⇒ Easy!



- If K ≠ root ⇒ Set No PRIORITY to $(-\infty)$

⇒ Ex: $\left(\frac{R}{2}\right) \Rightarrow \left(\frac{R}{-\infty}\right)$ ROOT

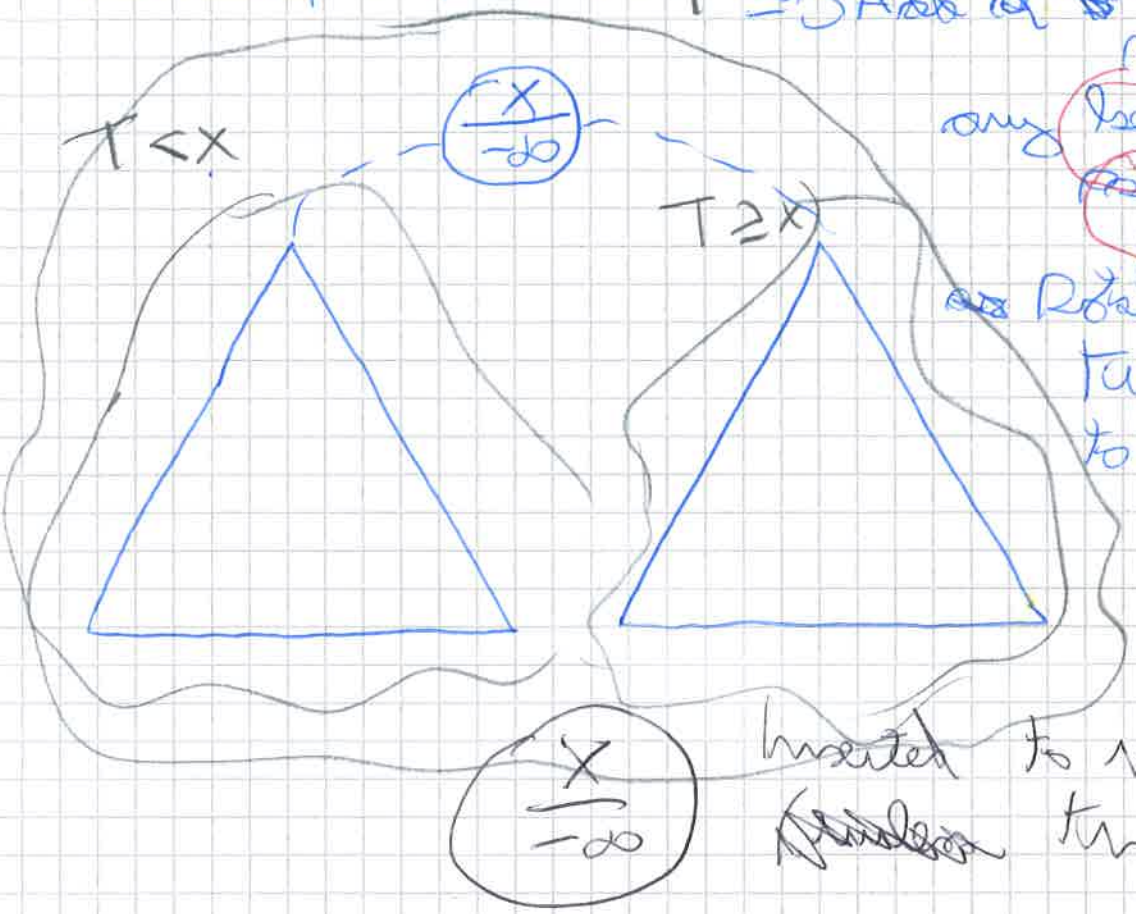
⇒ Push the node up to make it

- If K ∉ Treap:

  ↳ Insert $\left(\frac{K}{-\infty}\right)$, then perform SPLIT when it's at the root.

- MERGE (T_1, T_2) → O(depth)

T ⇒ Add a "DUMMY ROOT" with any key and priority = −∞ as Root and to keep the node up to the root.



RANDOMIZED TREAP.

A RANDOMIZED TREAP is a TREAP, where the priorities are assigned randomly to nodes. (uniformly and independently)

→ Since priorities are RANDOM, each node is equally likely to have the SMALLEST priority

FACT:

If the PRIORITIES are picked RANDOMLY, then the TREAP is BALANCED ⇒ DEPTH = $O(\log n)$

ie OPS. (insertion, deletion, search) take $O(\log n)$ time.

DEPTH!

PROOF: (like RAND. QUICKSORT)

KEYS: $\underbrace{x_1, x_2, \ldots, x_k}_{\text{INCREASING ORDER}}$ = Nodes with the kth-smallest search key.

We now define an INDICATOR R.V:

$A_k^{(i)} = 1$ iff: x_i is an ANCESTOR of x_k.

(ancestor, i; node k)

$$\text{depth}(x_k) = \sum_{i=1}^{n} A_k^i$$

(# ANCESTORS preceding node k)

Take $E\{\cdot\}$ of both sides:

$$E\{\text{depth}(x_k)\} = \sum_{i=1}^{n} E\{A_k^i\}$$

$$A_k^i = \begin{cases} 1 & \text{iff } x_i \text{ is ANCESTOR of } x_k \\ 0 & \text{otherwise} \end{cases}$$

$$E\{\text{depth}(X_k)\} = \sum_{i=1}^{n} E\{A_k^i\} =$$

$$= \sum_{i=1}^{n} 1 \cdot P\{A_k^i = 1\} + 0 \cdot P\{A_k^i = 0\}$$

$$= \sum_{i=1}^{n} P\{A_k^i = 1\}$$ (Given two keys, when is one key an ancestor of the other one?)

OBSERVATION:

Let's take: $\{X_1, \dots, X_{i-1}, X_i, \dots, X_k, X_{k+1}, \dots, X_n\}$

~~Given that priorities are assigned randomly,~~ when is X_i an ANCESTOR of X_k?

Firstly, we notice that:

$\{X_1, \dots, X_{i-1}, X_i, \dots, X_k, X_{k+1}, \dots, X_n\}$

$X_1, \dots, X_{i-1}$: MOVE IN BLOCKS & DO NOT INFLUENCE the case.

$X_i, \dots, X_k$: $|k-i|+1$

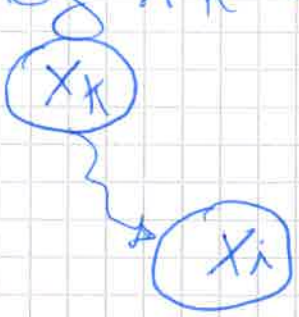
$X_{k+1}, \dots, X_n$: MOVE IN BLOCKS & DO NOT INFLUENCE the case.

$\Rightarrow$ We have 3 cases (we will pick the one where X_i is an ancestor of X_k)

(a) Priority of X_i is the MINIMUM one.

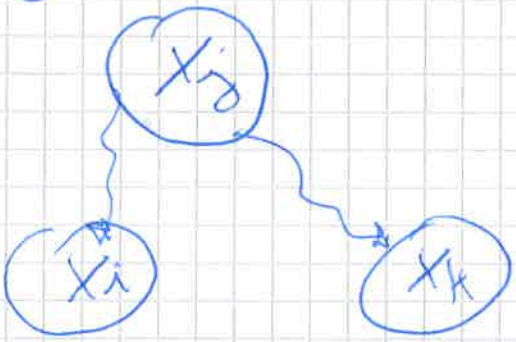
$\Rightarrow$ X_i is the root & surely an ancestor of X_k.

(2) PRIORITY of X_k is the MINIMUM one:



X_k is the root & an ancestor of X_i by

(3) PRIORITY of X_j is the MINIMUM one.



$\Rightarrow$ X_i can't be an ancestor of X_k, as X_i and X_k have gone to two different branches.

$\Rightarrow$ We are interested in SITUATION (1).
What is the P. of that case the request?

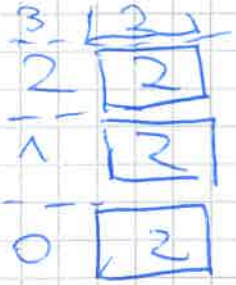
$$\sum_{i=1}^{n} P\{A_k^i = 1\} = \sum_{i=1}^{n} \frac{1}{|k-i|+1} = O(\log n)$$

Q.E.D.

$P\{$ Priority$(X_i) <$ Priority$(X_k)\}$

$X_1 X_2 \ldots X_i X_{i+1} \ldots X_{k-1} X_k X_{k+1} \ldots X_n$

$\Rightarrow$ We have hence proved that TREAPS with random priority are balanced & their depth is $O(\log n)$



$\Rightarrow \underbrace{H H \ldots H}_{l \text{ heads}}$

$\Rightarrow$ to be at level (l), we need to have atleast at least l HEADS!

$$= \sum_{i=1}^{n} \frac{1}{2^i} \left(P\{H\}\right)^l = \sum_{i=1}^{n} \frac{1}{2^i} \cdot \left(\frac{1}{2}\right)^l$$

$$= \left(\frac{1}{2}\right)^l \cdot \sum_{i=1}^{n} (1) = \frac{n}{2^l} \quad (\text{if } l < \log n \text{ is certain})$$

$\rightarrow$ height

$$\lg l = c \cdot \log n$$

$$P\{L \geq c \cdot \log n\} \leq \frac{n}{2^l} = \frac{n}{2^{c \log n}}$$

L: WEIGHT of SKIP LIST

$$= \frac{n}{2^{\log n^c}} = \frac{n}{n^c}$$

$$= \frac{1}{n^{c-1}} \leq \frac{1}{n}$$

$$\lg c \geq 2$$

$$c.$$

$$\Rightarrow P\{L \geq 2 \cdot \log n\} \leq \frac{1}{n}$$

with high prob!

$\Rightarrow$ Decreasing by $\frac{1}{n}$ ratio!

$\Rightarrow$ We have proved that with high P, a SKIP LIST has $O(\log n)$ LEVELS

$E\{\#LEVELS\} = O(\log n)$ with high prob.

SEARCH(k) ⇒ Start from the TOP-MOST LIST at the LEFT-MOST CORNER.

⇒ Go as far as possible to the right, then go down

[going down, you increase the "detail"]

as: next(u) ≤ k

⇒ GO RIGHT

if next(u) > k

⇒ GO DOWN

TIME: ⇒ $O(\log n)$, based on the #LEVELS.

#LEVELS is $O(\log n)$ with HIGH PROB.

PROOF:

We need to prove that the depth (max. #LEVELS) is logarithmic to the # keys.

$P\{\#LEVELS \text{ in SKIP LIST} \geq l\} \leq \sum_{i=1}^{n} P\{L(i) \geq l\}$

MAX. #LEVELS

$P\{\text{MAX. \#LEVELS in SKIPLIST} \geq l\} \leq \sum_{i=1}^{n} P\{L(i) \geq l\}$

level of the i-th key

Ex: To reach level 3 at the i-th key we must have had at least 3 heads (and tosses) in a row.

3 [2]
2 [2] ⇒ H H ... H
1 [2] (l heads)
0 [2]

⇒ to be at level (l), we need to have gotten at least l HEADS!

$$= \sum_{i=1}^{n} \frac{1}{n} \left(P\{H\}\right)^{l} = \sum_{i=1}^{n} \frac{1}{n} \cdot \left(\frac{1}{2}\right)^{l}$$

$$= \left(\frac{1}{2}\right)^{l} \cdot \sum_{i=1}^{n} \frac{1}{n} = \frac{n}{2^{l}} \quad (\lg l < \log n \text{ is taken})$$

→ height

$$\lg l = c \cdot \log n$$

$$P\{L \geq c \cdot \log n\} \leq \frac{n}{2^{l}} = \frac{n}{2^{c \cdot \log n}}$$

L: HEIGHT of SKIP LIST

$$= \frac{n}{2^{\log(n)^{c}}} = \frac{n}{n^{c}}$$

$$= \frac{1}{n^{c-1}} \leq \frac{1}{n}$$

$\lg c \geq 2$

$c \geq 2$

$$\Rightarrow P\{L \geq 2 \cdot \log n\} \leq \frac{1}{n}$$

writer high prob!

⇒ Decreasing by $\frac{1}{n}$ ratio!

⇒ We have proved that writer high P, a SKIP LIST has $O(\log n)$ LEVELS.

COMPLEXITY OF SEARCHING

PROOF: (and of deleting / inserting on element)

SEARCH = # VERTICAL STEPS + # HORIZONTAL STEPS (GO UP + LEFT STEPS) (GO UP)

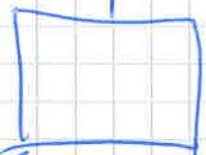
$E\{\text{SEARCH}\} = E\{\#\text{VERTICAL STEPS}\} + E\{\#\text{HORIZONTAL STEPS}\}$

We know that:

$E\{\#\text{VERTICAL STEPS}\} = L = O(\log n)$

(Height of the SKIP LIST)

We know that: $P\{\text{GO UP from a NODE}\} = \frac{1}{2}$



$P\{\text{GO UP from a NODE}\} = P\{\exists \text{ VERTICAL LINK from a NODE}\}$

$= P\{\text{H}\} = \frac{1}{2} \Rightarrow E\{\#\text{VERTICAL LINKS}\} = O(\log n)$

$P\{\text{GO LEFT from a NODE}\}$

$= P\{\nexists \text{ VERTICAL LINK from a node}\}$

$= P\{\text{T}\} = \frac{1}{2} \Rightarrow E\{\#\text{HORIZONTAL LINKS}\} = O(\log n)$

$\Rightarrow$ # NODES that have NO VERTICAL LINK

NODES that do have a VERTICAL LINK

$E\{\text{SEARCH}\} = E\{\#\text{VERTICAL STEPS}\} + E\{\#\text{HORIZONTAL STEPS}\}$

$= O(\log n)$ Q.E.D.

as the process of going up

is the process of going left

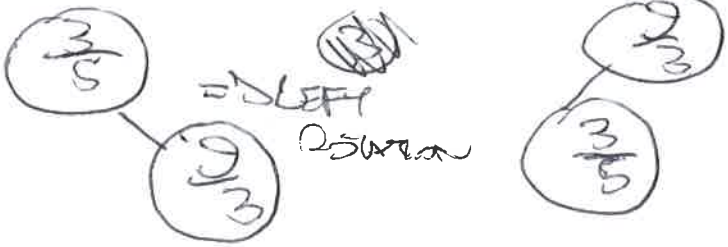
TREAP - CLASS EXERCISE:

S = { (3,5), (9,3), (15,9), (5,7), (10,7), (4,2), (2,11), (7,8) }

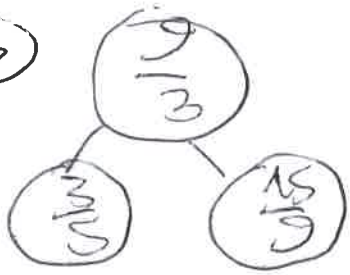
a) BUILD TREAP

1

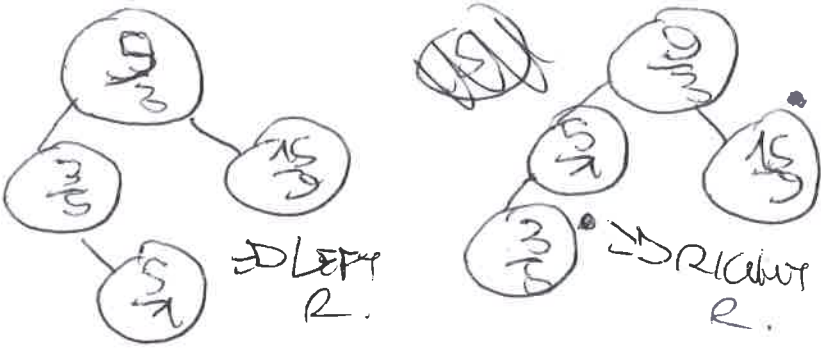
2



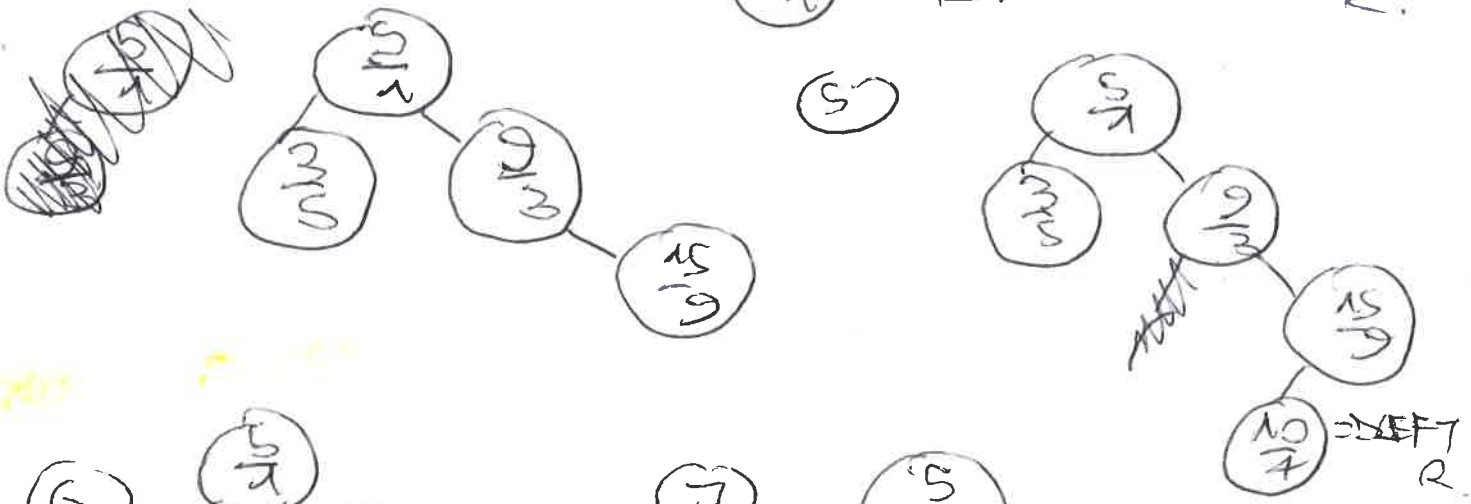
3



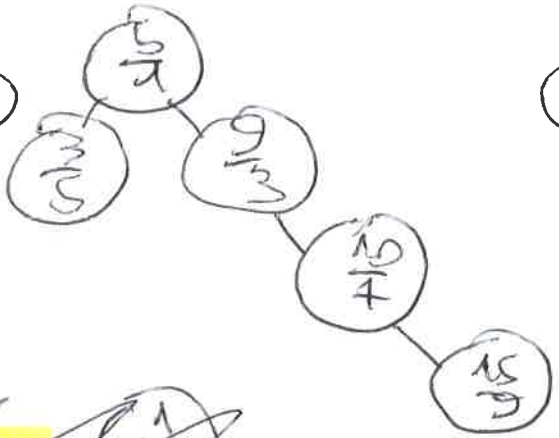
4



5

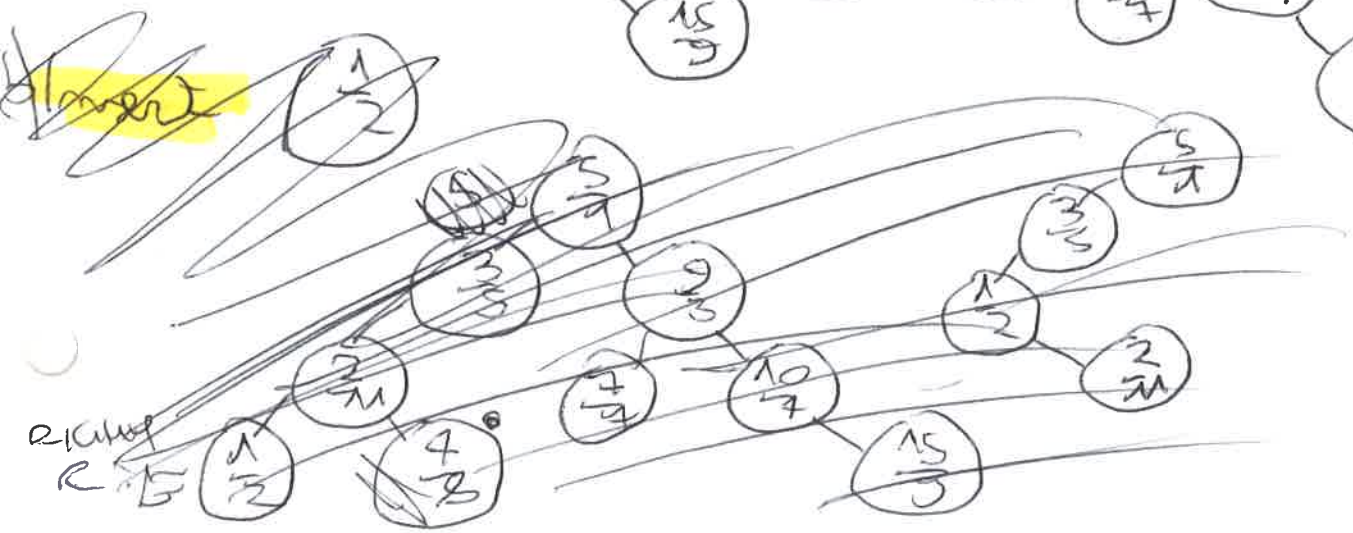


6



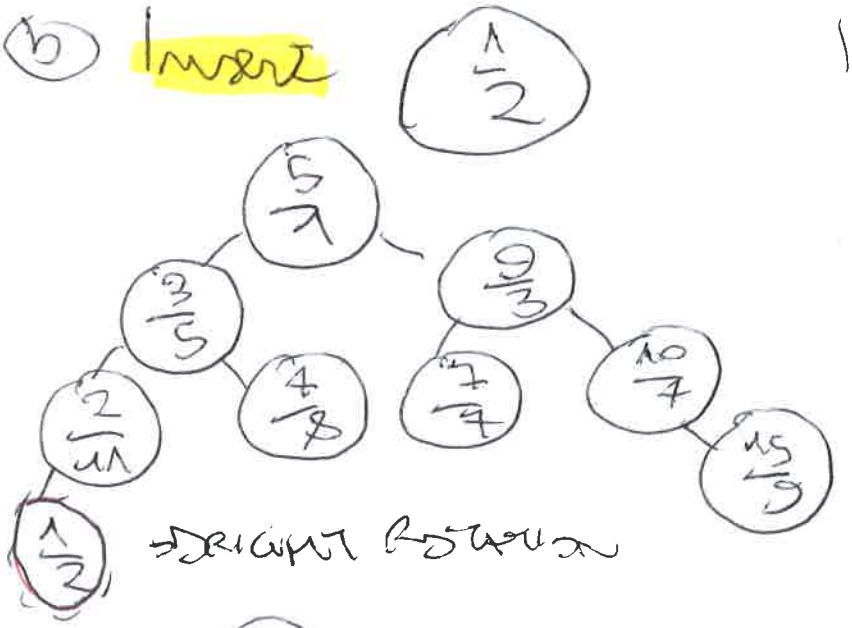
7

Insert

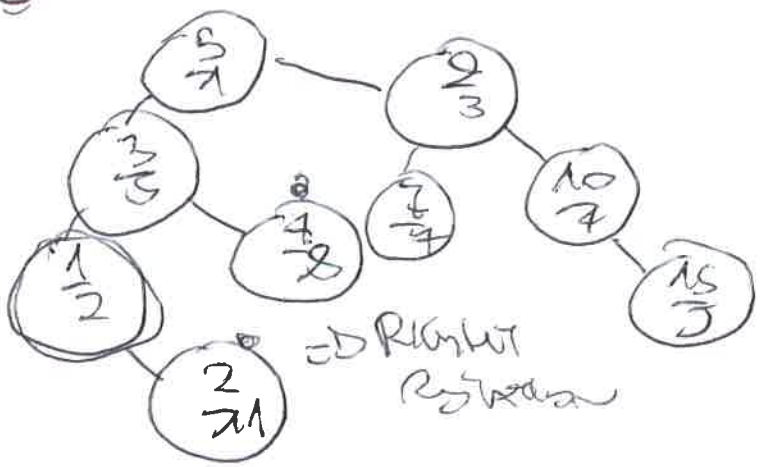


b) Insert

IDEA: Perform right left RIGHT ROTATION, then insert the remaining items (leaves) in the right position of the tree.



=> RIGHT ROTATION

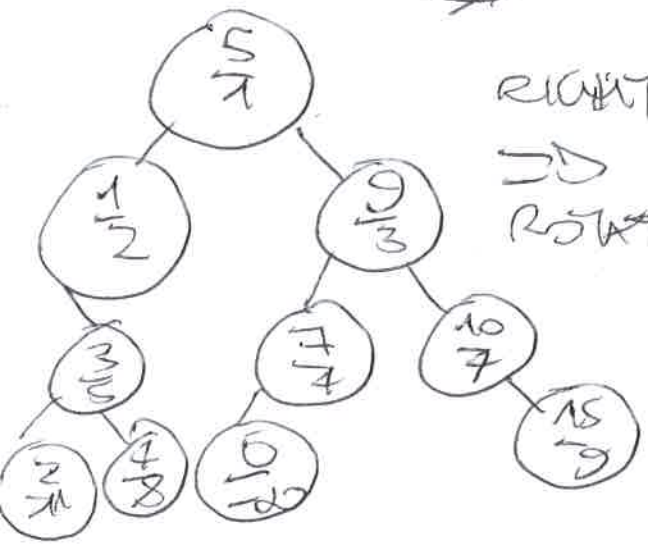


=> RIGHT ROTATION

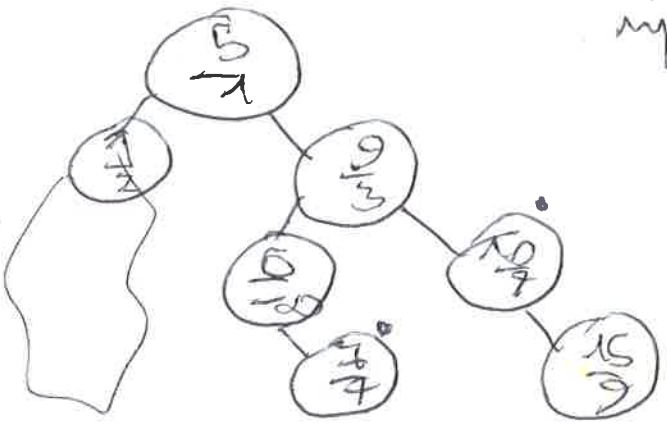
Okay!!

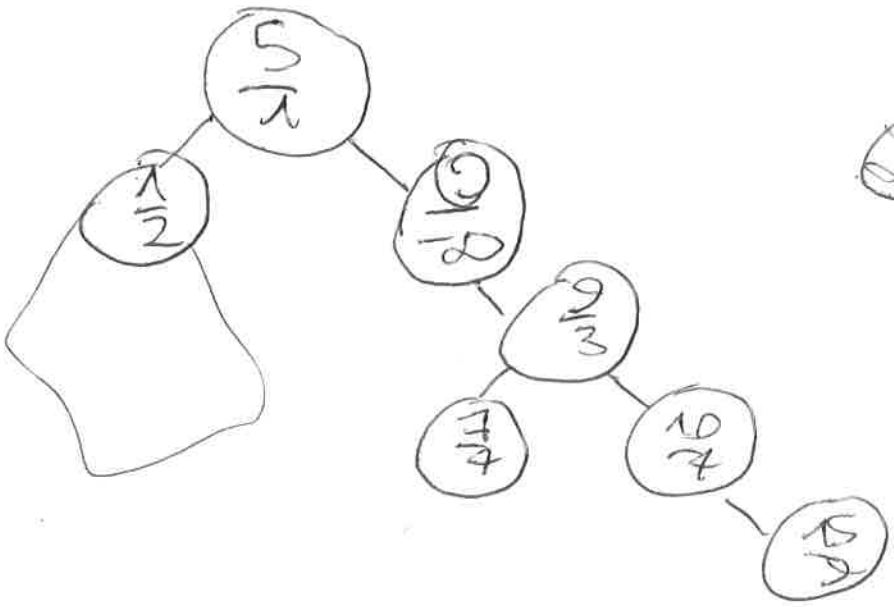
c) Need to perform a SPLIT at 6.

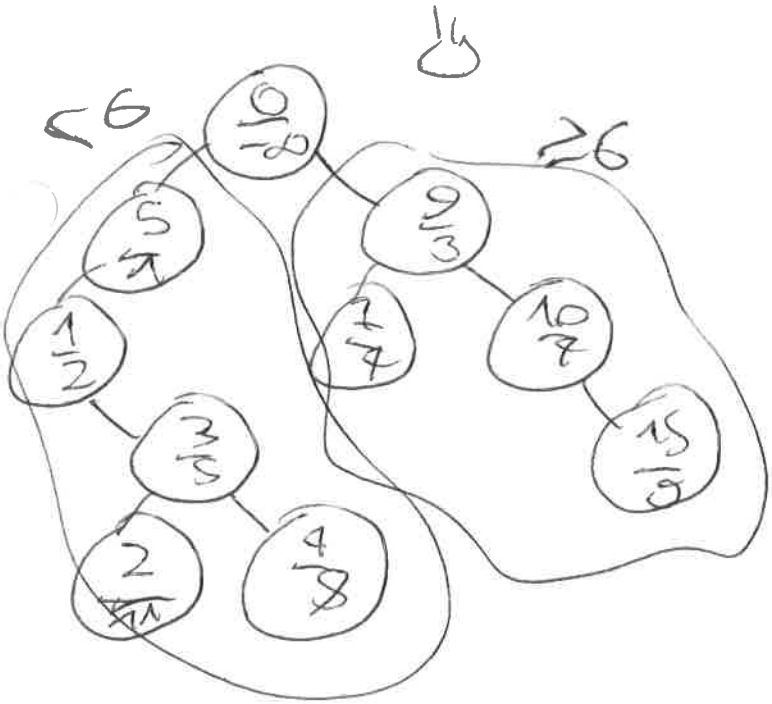
6 is inserted, and then "Right up".



RIGHT => ROTATION







DONE!