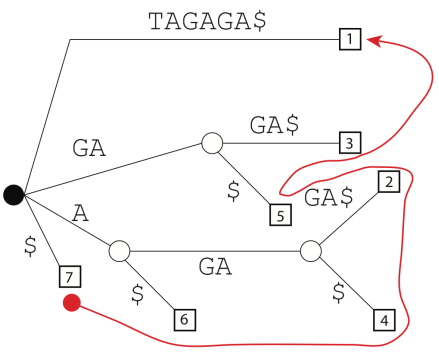
Algorithms and Data Structure for Population Scale Genomics

Lecture 1: Introduction

I. Nomenclature
   A. String: sequence of characters
   B. Alphabet: the set of characters that a string is constructed from
   C. Suffix: Given a string S, a suffix S_i is the substring of S that starts at position i and ends at the end of the string S
   D. Suffix Tree:
      1. Building a Suffix Tree
         a) Start with an empty root and a full string S
         b) Add successively each suffix, such that edge labels on path from root to leaf spell the suffix; label leaf with position
         c) Use existing edges where possible
         d) Contract edges that have no branches and concatenate labels
         e) Repeat for all edges
         f) The number of leaves does not change.
      2. Matching
         a) Start at root and traverse tree according to pattern
         b) Stop, if leaf or end of pattern is reached
         c) Find start positions of P in S in the leaves below of current sub-tree
         d) Search complexity is defined by the length of the pattern!
      3. Complexity
         a) Construction for string S of length n -> O(n)
         b) Search in S for a pattern P of length p -> O(p)
         c) Find matching statistics for P with p longest match between S and P -> $\Theta(p)$
   E. Suffix Array:
      1. Definition and Generation
         a) Sorted list of suffixes of a string S
         b) Generate list of suffixes, sort by lexicographic order
         c) Can be generated by a depth first traverse from the bottom of a suffix tree



Suffix Array

A = 7 6 4 2 5 3 1

d) Formal Definition: The suffix array $SA_T[1..n]$ of a text $T = t_1 t_2 \ldots t_n$ is the permutation of positions [1..n] such that $SA_T[i] = j$ iff suffix $T_{j..n}$ has position i in the list of all lexicographically sorted suffixes of T.

2. Matching
   a) Suffix Array A: A[k] is the start position of k-th smallest suffix.
   b) Search Pattern P: a substring of length p $\leq$ n
   c) Left Boundary L_p: $L_p = min(k : P \leq S_{A[k]} \; or \; k = n + 1)$
      (1) Smallest suffix that is larger or equal to pattern P
   d) Right Boundary $R_p = max(k : S_{A[k]} < Q \; or \; k = 0)$, where Q = P#, with # being a sentinel character lexicographically larger than any other in S or P
      (1) Largest suffix that is smaller than Q
   e) Using binary search to find L_p and R_p.
3. Complexity
   a) Construction and storage of Suffix Array A is $O(|S|)$
   b) Binary Search and pattern matching in $O(|P| \log |S|)$

II. Sequencing techniques
   A. Sanger sequencing
      1. Very low throughput
      2. Very low error rate
      3. Read length between 500 and 1000
      4. Often used in lab to verify SHORT sequences
   B. Illumina sequencing
      1. Very high throughput
      2. Low error rate
      3. Read length between 50 to 150
      4. Wildly used in the field of genomics and clinical sequencing
   C. PacBio
      1. Single molecule sequencing (no amplification)
      2. Length up to 1M bases
      3. Quite affordable and small hardware size
      4. High error rate (15%)
      5. Throughput of up to 250Gb/h (Ultra high throughput)
   D. Continuing technological improvements make the sequences longer and more accurate every year!

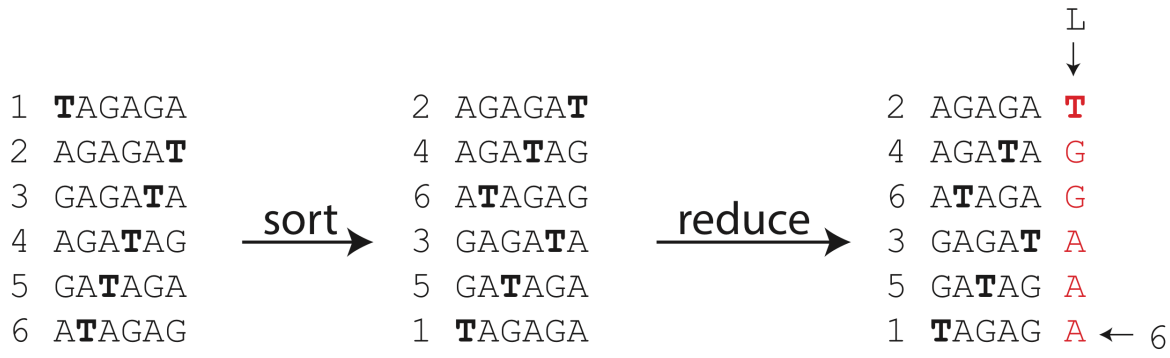
III. Motivation for large scale genomics analysis
   A. Advancement in measurement technology have transformed biology and medicine into areas of “big data”.
   B. Recorded sequences need to be processed using limited resources, such as laptops and smartphones.

IV. Example Question

A. When storing variants of a set of individuals in a PBWT, does this represent genotype or phenotype?
   1. genotype
B. Which technology would rather motivate large-scale genomics algorithms - illumine or Sanger sequencing? Why and How?
   1. Illumina sequencing. Because it has high throughput, which means, it is faster and more scalable than Sanger sequencing. It has the ability to produce large amount of sequencing data in a relatively short period of time.

Lecture 2: Suffix Array Construction

I. Suffix array matching
   A. Perform binary search to find $L_p \; and \; R_p$
      1. Search for L_p using pattern P
      2. Search for R_p using pattern Q
   B. Complexity: $O(|P|log|S|)$
II. Burrows Wheeler transform
   A. Construction
      1. Start with the full text T
      2. Generate all rotations of that text T
      3. Sort the strings lexicographically
      4. Keep only the last column L and the index of the original string



[B

   B. Useful Properties
      1. String L shows the long stretches of the same symbol
         a) Always have “GG” or “AAA”s in L
         b) String L has low entropy meaning it has long runs of the same character -> allows for compression
      2. Allows for very efficient compression
         a) Why? Long runs of the same character

b) Why? Certain characters are sticked together more often than others. Since we sort the first character by its lexicographic order, we partially sort its neighbours meaning the partially sort the last column of the rotation. If some characters are sticked together more often than others, we have a long run of the same character.

3. String L and index of original string are sufficient to reconstruct the original string
4. After sorting, indices are equivalent to suffix array.

III. FM-index

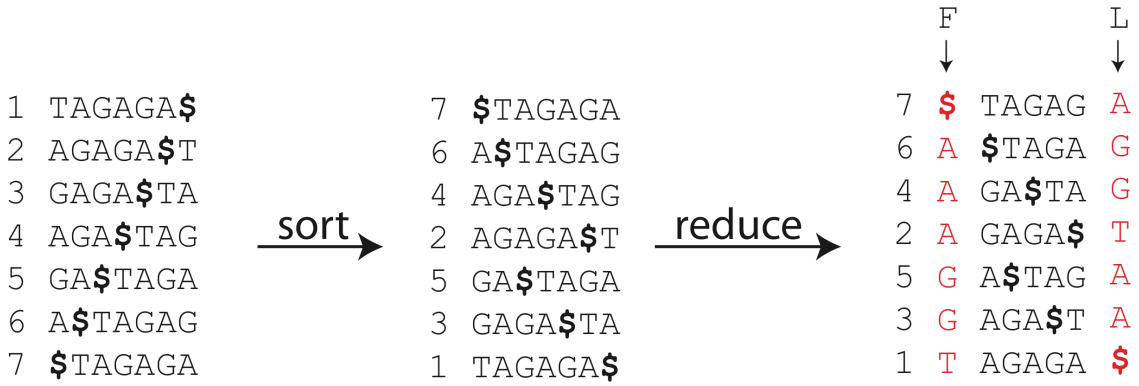
A. Full name: Full-Text Minute-Space index / Ferragina -Manzini index

B. Motivation

1. Suffix Array -> efficient search & Burrows-Wheeler Transform -> efficient compression.
2. Combine them for efficient search on compressed genome.

C. Construction

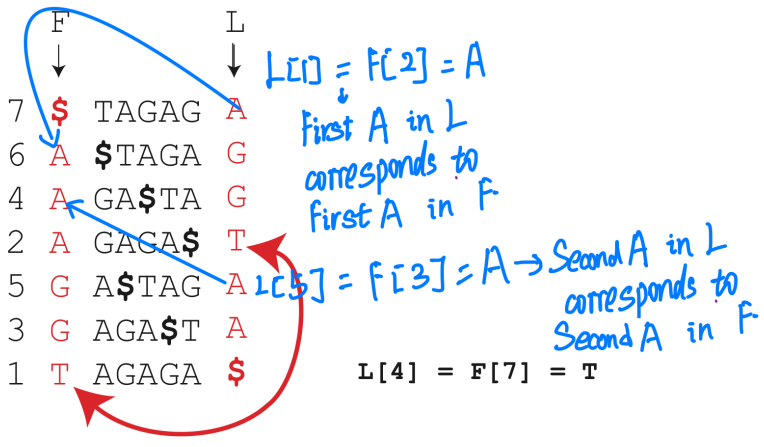
1. Add “$” to the end of the genome (“$” is smaller than all the alphabet the genome can take.)



2. Apply Burrows-Wheeler Transformation to string

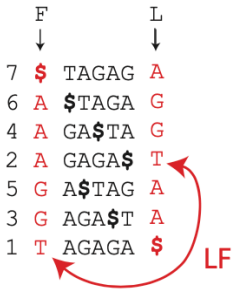
D. Last-to-First Column Mapping (LF Mapping)

1. The k-th occurrence on the last column L corresponds to the k-th occurrence on the first column F



E. Pattern Matching with FM-index

1. Auxiliary Structure to compute LF
   a) C: C[k] = total number of occurrence of characters < c
   b) Occ(c, k) = number of times c occurs in L[1, k]
      (1) L[1,k] substring from 1 to k
   c) Compute Last-to-First Mapping LF
      (1) LF(i) = C[L(i)] + Occ[L[i], i])
         (a) C[L(i)]: Starting position of character L(i)
      (2) Occ[L[i], i]): Up until L[1, i], how many character L(i) have occurred



Array C

$	A	G	T
0	1	4	6

Matrix Occ

	1	2	3	4	5	6	7
$	0	0	0	0	0	0	1
A	1	1	1	1	2	3	3
G	0	1	2	2	2	2	2
T	0	0	0	1	1	1	1

```
LF(4) = C[L[4]] + Occ(L[4], 4)
LF(4) = C[T] + Occ(T, 4)
LF(4) = 6 + 1
LF(4) = 7
```

2. Pattern Matching with FM-index
   a) pseudocode
      (1) i <- p, c <- P[p]
      (2) sp <- C[c] + 1
      (3) ep <- C[c + 1] # c+1: the next character that is larger than c
      (4) While (sp ≤ep) and (i ≥ 2) do
      (5) c <- P[i-1]
      (6) sp <- C[c] + Occ(c, sp - 1) + 1
      (7) ep <- C[c] + Occ(c, ep)
      (8) i <- i-1
      (9) End While
   b) Example

Example

P = **AGA**, $p = |P| = 3$

A← **Init:**
$c \leftarrow P[3] =$ **A** 有几个比A小.
$sp \leftarrow C[\mathbf{A}] + 1 = 2$ → start position of A
$ep \leftarrow C[\mathbf{A}+1] = C[\mathbf{G}] = 4$ → end position of A

GA ← **Iteration 1:** 有几个比G小，即有几个A.
$c \leftarrow$ **G**
$sp \leftarrow C[\mathbf{G}] + Occ[\mathbf{G}, 1] + 1 = 5$ 在sp上的有几个G
$ep \leftarrow C[\mathbf{G}] + Occ[\mathbf{G}, 4] = 6$ 在ep上的有几个G.

AGA← **Iteration 2:**
$c \leftarrow$ **A**
$sp \leftarrow C[\mathbf{A}] + Occ[\mathbf{A}, 4] + 1 = 3$
$ep \leftarrow C[\mathbf{A}] + Occ[\mathbf{A}, 6] = 4$

—sp
—ep

IV. Counting sort

A. Pseudocode

1. Given: a sequence (of characters / integers) L[1…n]
2. Task: Generate a sorted sequence L', such that $L'[i] \leq L'[i+1] \quad \forall i$
3. Collect character counts in array $count[1...\sigma]$ in O(n):

$$count[c] = |\{L[i] \,|\, L[i] = c, \forall i \in [1..n]\}|$$

4. Collect cumulative counts in array $B[1..\sigma]$ in $O(\sigma)$:

$$B[1] = 1 \quad \text{and} \quad B[k] = B[k-1] + count[k-1] \forall k \in [2..\sigma]$$

5. Compute L' in $O(n)$ time:

$$L'[B[L[i]]] = L[i] \quad \text{and} \quad B[L[i]] += 1$$

B. Complexity: $O(n+\sigma)$ time and space

C. Example:

Given: a sequence (of characters / integers) $L[1..n]$ with $L[i] \in [1..\sigma]$.

L=TAGAGA

1. collect character counts in array count$[1..\sigma]$ in $O(n)$:

count[c] = $|\{L[i] \mid L[i] = c, \quad \forall i \in [1..n]\}|$.

count = { 3 , 0 , 2 , 1 }
A C G T

2. collect cumulative counts in array $B[1..\sigma]$ in $O(\sigma)$:

$B[1] = 1$ and $B[k] = B[k-1] + \text{count}[k-1] \quad \forall k \in [2..\sigma]$

B = { 1 4 4 6 } ⇒ starting position of A, C, G, and T.
A C G T

3. compute L' in $O(n)$ time:

$L'[B[L[i]]] = L[i]$ and $B[L[i]] += 1$

L' = { A A A G G T }
1 2 3 4 5 6

B = { 3 4 6 7 }
A C G T

D. Idea of counting sort: construct the starting position for each of the character in Σ (stored in list B). Then whenever we need to sort, we just fill the character L[i] according to where it starts (B[L[I]]). Finally we increase the starting position of that character by 1. ($B[L[i]] += 1$)

E. Radix Sort:

1. Fill a list of alphabetic order from the last digit to the first digit
2. Need to scan the input #of largest digits time

V. Fixed-length string sorting

A. Sorting a list of key-value-pairs

1. Pseudocode
   a) Given: a list L of key-value pairs (L[I].p, L[I].v)
   b) Task: Generate sorted sequence L'
   c) Generate lists $count[1..\sigma]$ and $B[1..\sigma]$ as before using $L[i].p$
   d) Compute L' as follows:

$$L'[B[L[i].p]] = L[i], \ \forall i \in [1..n]$$
$$B[L[i].p] += 1$$

2. Complexity: O(n + σ) time and space
3. Idea: the idea is the same as counting sort.

B. Sorting a list of 2-key-value-pairs

1. Pseudocode
   a) Generate sorted lists $L^p \ and \ L^s$, sorted by primary and secondary keys, respectively using the key-value-pair based sort in $O(n+\sigma)$ time
   b) For the computation of L^P stop at the cumulative counts B^p
   c) Then fill L' with values from L^s, using offsets from B^p

$$L'[B^p[L^s[k].p]] = L^s[k], \forall k \in [1..n]$$
$$B^p[L^s[k].p] += 1$$

2. Complexity: $O(n+\sigma)$ time and space
3. Idea: This algorithm exploited the idea of Radix Sort, first we sort the 2-key-value pair by the secondary key, and then by the primary key.

C. Sorting a list of fixed-length strings

1. Task: Given a list of strings $S = \{W^1, W^2, \ldots, W^n\}$ with $|W^i| = k$. Generate the lexicographically sorted list S^*
2. Algorithm:
   a) Use the 2-key sort algorithm to generate a sorted list L', from list L of triples (p, s, v) that represents all length-2 prefixes:

$$L = (w_1^1, w_2^1, 1), (w_1^2, w_2^2, 2), \ldots, (w_1^n, w_2^n, n)$$

   (1) w_1^i means the first character in word i
   b) Form list R[1..n] that stores the lexicographic ranks of $L'[1..n]$, storing the number of distinct 2-key pairs in $L'[1..j]$

$$R[j] = \begin{cases} R[j-1] & \text{if key-pair at } L' = \text{key-pair at } L'[j-1] \\ R[j-1]+1 & \text{otherwise} \end{cases}$$

   c) Form a new list L using the values from R as primary keys, that now represents the length-3 prefixes of all strings:

$$L = (R[1], w_3^{L'[1].v}, L'[1].v), (R[2], w_3^{L'[2].v}, L'[2].v), \ldots, (R[n], w_3^{L'[n].v}, L'[n].v)$$

d) Generate a new sorted list L' in $O(n+\sigma)$ and iterate steps 2-4 k - 2 in total. After k iterations $S^*[j] = W^{L[j].v}$

3. **Complexity:** $O(k(n+\sigma))$
4. Idea: First sort the first 2-length prefixes using the 2-key-pair sorting. Then form a list R that stores the lexicographic ranks of each distinct 2-key pairs. In the next steps, the R is a list stored to represent the orders of the length x prefixes. Then we use the second list L to extend the prefix length by 1, so that we have the sorted order of x+1-length prefix. Iterate k-2 times, we have the sorted k-length prefix.

VI. Example Question

A. Why is the complexity of exactly matching a pattern P against a suffix array of a text T in worst case $O(log|T||P|)$

1. Because, we performed two binary search on sorted suffixes to find the left and right boundary of pattern P, this would take $O(log|T|)$ many comparisons . Then when doing the binary search, we need to compare the pattern P to some suffix, this in the worst case would take O(|P|) for each comparison. So we have $O(log|T||P|)$

VII. Summary

A. In this lecture we covered different integer and string sorting algorithms

B. How to

1. Sort integers using counting sort in $O(n+\sigma)$
2. Sort lists by primary keys and primary-secondary key pairs in $O(n+\sigma)$
3. Sort strings of a **fixed** length k in $O(k(n+\sigma))$

Lecture 3: Index construction in linear time

I. Variable length string sorting

A. We are given a list S of n variable-length strings $W^1, W^2, \ldots, W^n$, with $W^i \in [1..\sigma]^*$: Let m denote the length of the longest string:

$$m = \max\{|W^k| : k \in [1..n]\}$$

B. Further, let S^* be the list of all strings in S in lexicographic order. We also define S^p as the set of all prefixes of length p in S:

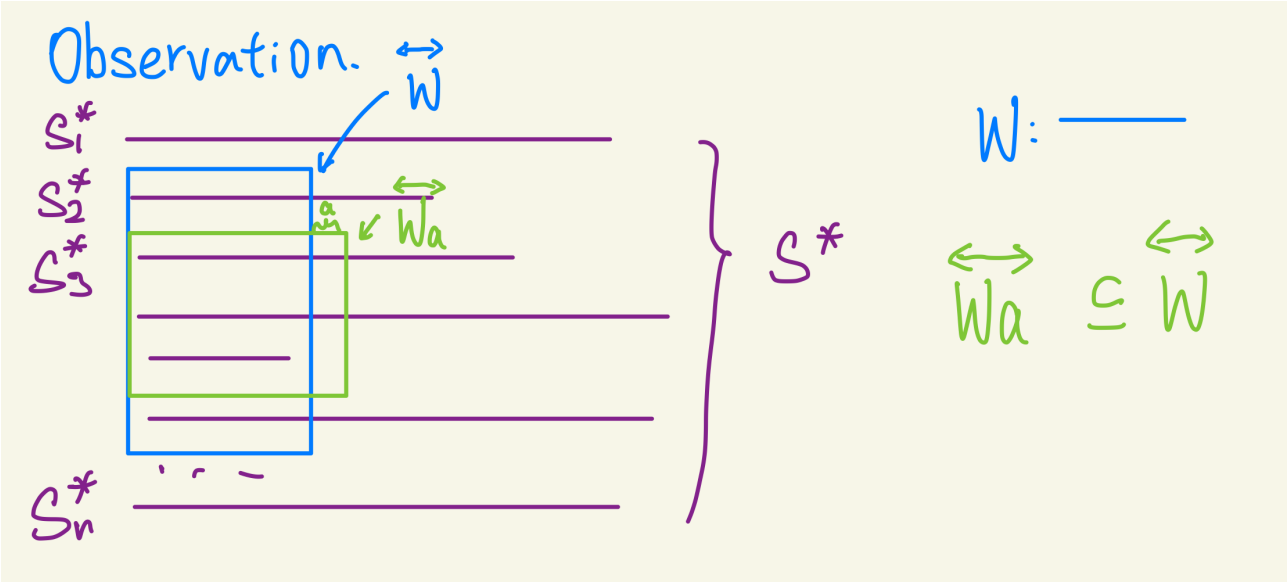
$$S^p = \{W^k_{1..p} | k \in [1..n]\}$$

C. Last, let us define that a prefix can never include more than the whole string

$$W^k_{1..p} = W^k, \text{ if p} > |W^k|$$

D. Observation:

1. Every string $W \in S^p$ maps to a contiguous interval $\overleftrightarrow{W} = [\overleftarrow{W} .. \overrightarrow{W}]$ in S^* that contains all strings starting with W
2. Consequences
   a) Prefixes of length p imply a partitioning I^p of S^*:
   $I^p = \{\overleftrightarrow{W} \mid W \in S^p\}$
   b) A partition I^P is a refinement of partition I^{p-1} as $\overleftrightarrow{Wa} \subseteq \overleftrightarrow{W}$



E. Sorting algorithm

1. preparation
   a) Given a list of n strings of different lengths, we can lexicographically sort it into array S^* in $O(N + \sigma)$ time where
   $N = \sum_{i=1}^{n} |W^i|$
   b) Let L be the list of **all possible** triplets (pos, char, str) where
      (1) pos: the current position in string str
      (2) char: the character at current position $W^{str}[pos]$
      (3) str: the index of the current string
      (4) Using result from lecture 2, we can sort L using pos and char as primary and secondary keys, respectively, in $O(N)$ time

2. Sorting Algorithm
   a) Given the sorted list of triplets L', let L^p describe the continuous interval in L' containing all triplets with primary key p(pos)

b) The following algorithm will contain an initialisation round and iterations of an update/refinement round. Given that the longest word in S has length m, we need in total m iterations, so $p \in [1..m]$

c) Initialisation:(p = 1)

(1) Generate a block pointer array B[1..n] and initialise it as

$$B[k] = |\{h : W_1^h < W_1^k\}| + 1$$

This reflects the partitioning I^1 for all prefixes of length 1

d) Update/Refinement:(p > 1)

(1) Assume that $B[j]$ contains the starting positions $\overleftarrow{W}^j_{1..p-1}$ of the continuous interval implied by the prefix string $W^j_{1..p-1}$ in S^*

(2) We define the following auxiliary structures:

(a) Q[1..n] : current position inside an interval, $Q[\overleftarrow{T}] = \overleftarrow{T}a$

(b) S[1..n] : current size of each interval $S[\overleftarrow{T}] = |\overleftrightarrow{T}|$

(c) C[1..n] : current character active at each interval $C[\overleftarrow{T}] = a$

(d) $\overleftarrow{T}_k = B[L^p[k].str]$: the start position of the interval prefixed by $W^{L^p[k].str}_{1..p-1}$ (k iterates over all active strings, that is members of L^p.

(3) Initialise Q in $O(|L^p|)$ time using two scans over L^p

(a) $\forall k : set \;\; Q[\overleftarrow{T}_k] = 0$

(b) $\forall k : if \;\; Q[\overleftarrow{T}_k] = 0$:

i) $Q[\overleftarrow{T}_k] = B[L^p[k].str] = \overleftarrow{T}_k$

ii) $C[\overleftarrow{T}_k] = L^p[k].char$

(4) Update B ($\forall k \in [1..|L^p|]$):

(a) if $C[\overleftarrow{T}_k] = L^p[k].char$:

i) $B[L^p[k].str] = Q[\overleftarrow{T}_k]$

ii) $S[\overleftarrow{T}_k] = S[\overleftarrow{T}_k] + 1$

(b) else:

i) $Q[\overleftarrow{T}_k] += S[\overleftarrow{T}_k] \quad S[\overleftarrow{T}_k] = 1$

ii) $C[\overleftarrow{T}_k] = L^p[k].char \quad B[L^p[k].str] = Q[\overleftarrow{T}_k]$

e) Complexity

(1) The initialisation step can be completed in O(n) time

(2) For the refinement/update step, we need for each iteration as many steps as there are characters active at this position. As

never any character is active more than once, the total number of steps equals the total number of characters in the input text

$$N = \sum_{i \in [1..n]} |W^i|$$

Hence, the sorting takes $O(N + \sigma)$ time

3. Idea: First we initialise block pointer B with the first character of each string. Then we update B so that it can represent the order of the length 2 prefixes. Update B so that is can represent the order of length 3 prefixes, and eventually we have the order of length m prefixes.
   a) Understanding the refinement step: The refinement step is describing a procedure that given the blue block from figure above, finding the position of the green block. Q is the starting position of the new green block. S is the size of the green block. Active character is marked by the C. If $C[\overleftarrow{T}_k] = L^p[k].char$ is met, we know that the character $L^p[k].char$ is still the same as the active character, so we increase the green block size by 1. If the condition is violated, we know that the character is no longer active and the green block is done increasing, we need to look at a new green block which means a new active character. So, we update Q about the starting position of the new green block, change the new green block size to 1, update the active character to the new active character, and finally, change the block pointer B accordingly.

II. Suffix array construction

A. small and large suffixes

1. We call a suffix $T_{i..n}$ of a string T **large** if $T_{i..n}$ is lexicographically larger than $T_{i+1..n}$. Analog, a suffix $T_{i..n}$ that is lexicographically smaller than $T_{i+1..n}$ is called small.
2. Lemma 3.2: Let $\overleftrightarrow{c}$ be the interval of single-character prefix $c \in [1..\sigma]$ in SA_T. In $\overleftrightarrow{c}$, large suffixes occur before small suffixes for any $c \in [1..\sigma]$
   a) Proof: Assume $T_{i..n}$ and $T_{j..n}$ are two suffixes of T, with $t_i = t_j = c$. Let $T_{i..n}$ be a small suffix and $T_{j..n}$ a large suffix. Then it follows that

$$T_{i..n} = c^p dv\#, \text{ with } p \geq 1, d \in [c+1..\sigma], v \in \Sigma^*$$
$$T_{j..n} = c^q bw\#, \text{ with } q \geq 1, b < c, w \in \Sigma^*$$

If $p < q$, then $T_{j..n} = c^p c^{q-p} bw\#$ and $T_{i..n} = c^p dv\#$, since d > c, then $T_{i..n} > T_{j..n}$.

if p > q, then $T_{i..n} = c^q c^{p-q} dv\#$ and $T_{j..n} = c^q bw\#$, since c > b, then $T_{i..n} > T_{j..n}$

if p = q, then $T_{i..n} = c^p dv\#$ and $T_{j..n} = c^p bw\#$, since d > b, then $T_{i..n} > T_{j..n}$.

Hence, $T_{i..n} > T_{j..n}$.

B. Creating array of small suffixes
   1. Given text T of length n, we can label all start positions k of any suffix as being small or large with a right-to-left scan in O(n) time.
      a) Consider a suffix T_i (i < n)
      b) If $t_i \neq t_{i+1}$, we only need to compare t_i and t_{i+1} to determine if T_i is of small suffix or large suffix
      c) if $t_i = t_{i+1}$, find the smallest $j > i$ such that $t_j \neq t_i$
         (1) if $t_j > t_i$, then suffixes $T_i, T_{i+1}, \ldots, T_{j-1}$ are small suffixes
         (2) if $t_j < t_i$, then suffixes $T_i, T_{i+1}, \ldots, T_{j-1}$ are large suffixes
   2. From this effort, we concrete an array $I[1..m] = [i_1, i_2, \ldots, i_m]$ that contains the positions of all small suffixes in T.
   3. Utilising I, we can now initialise an incomplete version of the suffix array SA_T, which we call $\widetilde{SA_T}$, that contains the same entries as SA for all positions $i \in I$ and 0 otherwise
   4. Following we will show how to generate SA_T from $\widetilde{SA_T}$ in O(n) time.
      a) Assume that the entries of $\widetilde{SA_T}[1..k]$ are already filled and we are currently at position k. Now consider the previous suffix $T[\widetilde{SA_T}[k] - 1..n]$ and set $c = T[\widetilde{SA_T}[k] - 1]$
      b) This generate two possibilities
         (1) $T[\widetilde{SA_T}[k] - 1..n] < T[\widetilde{SA_T}[k]\,..\,n]$ -> nothing to do since $T[\widetilde{SA_T}[k] - 1..n]$ is also a small suffix
         (2) $T[\widetilde{SA_T}[k] - 1..n] > T[\widetilde{SA_T}[k]\,..\,n]$ -> append $T[\widetilde{SA_T}[k] - 1..n]$ to the current list of suffixes starting with c
         (3) Since we do this as we iterate through $\widetilde{SA_T}$ and thus through a sorted list of suffixes
         (4) Complexity: O(n)

C. Initialisation of $\widetilde{SA_T}$

1. Let $I = i_1, i_2, \ldots, i_m$ be the list of starting positions of all small suffixes in T. Further let S be the list of all suffixes $T[i_1 \ldots n], \ldots, T[i_m \ldots n]$ in T and S^* its permutation in lexicographic order
2. Let vector R store at R[k] the position of suffix $T[i_k \ldots n]$ in S^* and let $R[m+1] = \#$, Assume we can generate suffix array SA_R from R interpreted as a string
3. Given SA_R, we can follow a simple procedure to fill $\widetilde{SA_T}$ at all positions $i \in I$:
   a) Compute block pointer array $Q[1..\sigma]$ that gives the starting position of each character block in SA_T from a single scan over T.
   b) Let u_c be the number of large suffixes in T starting with c. Then update $Q[c] += u_c$
   c) Using lemma 3.2 and $t_k = T[I[SA_R[k]]]$, we can fill $\widetilde{SA_T}$ iterating over SA_R with $k \in [2..m+1]$

$$\widetilde{SA_T}[Q[t_k]] = I[SA_R[k]]$$
$$Q[t_k] += 1$$

D. Computing SA_R

1. Generate a list of strings $W = T[i_1 \ldots i_2]\$, T[i_2 \ldots i_3]\$, \ldots, T[i_m \ldots n]\$$ describing the string differences between all neighbouring small suffixes in text T.
2. As the total length of W is O(n), we can apply sorting algorithm to generate a vector W in O(n) time that presents at $W[k]$ the position of $T[i_k \ldots i_{k+1}]\$$ in the lexicographically sorted list W^*
3. With setting $W[m+1] = \#$, we can again generate a suffix array SA_w
4. Lemma 3.3 : For any two suffixes $T[i_k \ldots n]$, $T[i_h \ldots n]$ with $i_k, i_h \in I$, it hods that if i_k follows i_h in SA_w, then i_k follows i_h also in SA_R and hence $SA_W = SA_R$
   a) Proof: Assume that i_k follows i_h in SA_w, then we can distinguish the following cases:
      (1) $S\$ = T[i_k \ldots i_{k+1}]\$ \neq T[i_h \ldots i_{h+1}]\$ = S'\$$
         (a) a prefix $S_{1..p}$ with $p < |S|$, is not prefix of S' -> $T[i_k \ldots n] > T[i_h \ldots n]$
         (b) S is a prefix of S', but S$ is not

i) $T[i_k..n] = VT[i_{k+1}..n]$ with $S = Vc$, $V \in \Sigma^*$

ii) $T[i_h..n] = VT[i_{h+|S|-1}..n]$

iii) $T[i_{k+1}] = T[i_{h+|S|-1}] = c$ but $T[i_{h+|S|-1}] \notin I$

(1) $T[i_{k+1}..n]$ is a small suffix

(2) $T[i_{h+|S|-1}..n]$ is a large suffix

iv) $T[i_{h+|S|-1}..n] < T[i_{k+1}..n]$ according to lemma 3.2

v) $T[i_k..n] > T[i_h..n]$

(2) $S\$ = T[i_k..i_{k+1}]\$ = T[i_h..i_{h+1}]\$ = S'\$$

(a) W[k]=W[h] and W[k+1..m]# > W[h+1..m]#

(b) $\exists x \geq 1 : T[i_{k+x}..i_{k+x+1}] > T[i_{h+x}..i_{h+x+1}]$

(c) $\forall y < x : T[i_{k+x}..i_{k+x+1}] = T[i_{h+x}..i_{h+x+1}]$

(d) $T[i_k..n] > T[i_h..n]$

b) Proof ends

5. To efficiently compute SA_w, we can apply the same mechanism recursively, until we reach an acceptable small constant size of w.
6. complexity can be computed as
   a) $f(n, \sigma) = an + b\sigma + f(\frac{n}{2}, \frac{n}{2}) \in O(n + \sigma)$
7. In case that the number of small suffixes in T is larger than $\frac{n}{2}$, we can use the same mechanism on large suffixes ,making the analog changes to our argumentation.

III. Exam Questions

A. What is the relation ship of small and large suffixes of a string and how can we get one from the other?
   1. Answer: large suffix is the suffix which is larger than its successor and small suffix is the suffix that is smaller than tis successor. The relationship of large and small suffixes is that the large suffix is always smaller than the small suffix if the two kinds of suffixes share the same starting character.

B. What is the time complexity of variable-length sorting and why?
   1. The time complexity of variable-length sorting is $O(N + \sigma)$, where N is the total number of characters and σ is the number of alphabet.
   2. First reason is we need to sort the triples using first and secondary key, and there are O(N) triples, so we need $O(N + \sigma)$ time for that.
   3. Then for the initialisation and update phase, we need to calculate the number of active character times, and since the number of active character is O(N), we need $O(N + \sigma)$ time.

Lecture 4: Burrows Wheeler Transform (BWT)

I. Construction of BWT

A. With out linear time construction of suffix arrays, we now have a tool in hand that we can utilise. The structure of BWT implies a relationship between $SA_{T\#}$ and $BWT_{T\#}$

$$BWT_{T_{\#}[i]} = \begin{cases} T[SA_{T_{\#}}[i] - 1] & \text{if } SA_{T\#}[i] > 1 \\ \# & \text{else} \end{cases}$$

II. Range Minimum Queries (RMQ)*

A. This is not an exam content but is used constantly in later courses

B. Binary search tree

1. A binary search tree data structure is a binary tree T that stores in each leaf v a (key(v), value(v)) pair and at each internal node v a pair (key(v^R), value(v)), where v^R is the right-most leaf in the sub-tree rooted by v and value(v) is application dependent
2. Leaves are ordered left to right by key()
3. If T is balanced, traversal from root to any one of the n leaves takes $O(log(n))$ steps
4. In addition to lookup of leave values, we can also implement range queries

C. Range Minimum Query (RMQ)

1. Given a binary search tree T, the operation $RMQ(l, r)$ returns the minimum element amongst the leaves w in a given key range:

$$RMQ(l, r) = \min_{w:l \leq key(x) \leq r} val(w)$$

2. We can compute $RMQ()$ in $O(logn)$ time

III. Rank and Select data structures

A. For navigating data structures such as BWT of a text T, we often need to locate an instance of a certain character or count how often it occurs up to a position i.

B. $rank_c(T, i)$: How many instances of character **c** did we see in text T up to position i

$rank_c(T, i)$ count # of c up to i

A	C	A	G	T	A	C	C	G	A	T
0	1	2	3	4	5	6	7	8	9	10

$$rank_A(T,5) = 3$$

C. $select_c(T,i)$: At which position in text T occurs character **c** for the **i**-th time

A	C	A	G	T	A	C	C	G	A	T
0	1	2	3	4	5	6	7	8	9	10

$select_c(T,i)$ get the i-th c

$select_G(T,2) = 8$

D. In $rank_c(T,i)$, i means **up until the i-th place in text T**

E. In $select_c(T,i)$, i means **the i-th occurrence of character c**

F. **Rank and Select for Binary vectors**

1. Assume we are given a binary vector $B[1..n]$ with $B[i] \in [0,1]$. We make the following definitions:
   a) Definition : Rank
      (1) The operation $rank_c(B,i)$ computes the number of occurrences of c up to and including position i

$$rank_c(B,i) = |\{i' | 1 \le i' \le i, B[i'] = c\}|$$

   b) Definition : Select
      (1) The operation $select_c(B,i)$ computes the index of the i-th occurrence of c in B

$$select_c(B,i) = j, \text{ with } rank_c(B,j) = i$$

2. Efficient rank and select
   a) Goal: find a good time-space trade-off, we can easily get constant time rank using $O(nlogn)$ space, if we pre-compute and store all ranks. (we have to store n numbers and each number takes up $logn$ bits)
   b) Store the pre-computed rank only for a subset of i/ℓ positions:

$$first[\lfloor i/\ell \rfloor] = rank(B,i)$$

For $\ell = (\lceil (logn)/2 \rceil)^2$: first takes $O(n/logn)$ space.
We would then calculate rank as:

$$rank(B,j) = first[j/\ell] + count(B[\ell * (j/\ell) + 1..j])$$

Where $count(B[i..j])$ counts the number of set bits in B[i..j]
Which takes $O(log^2 n)$ time worst case since we would have to walk through the whole block.

c) Further compartmentalise the ℓ-length blocks and store blocks of length k:

$$second[\lfloor i/k \rfloor] = count(B[\ell * (i/\ell) + 1..i])$$

With choosing $k = \lceil (logn/2) \rceil$, this costs $O(n/k * log\ell)$ bits, that is $O(nloglogn/logn)$ space.

Now rank can be computed in $O(logn)$ time:

$$rank(B, i) = first[i/\ell] + second[i/k] + count(B[k * (i/k) + 1..i])$$

d) Further update idea: With a block length of $k = \lceil (logn/2) \rceil$, the number of possible bit-vectors is upper bounded by $\sqrt{n}$ ($2^k = 2^{logn/2} = (2^{logn})^{\frac{1}{2}} = \sqrt{n}$). We can directly encode all possible rank values

(1) We form a table $third[0..2^{k-1} - 1]$ that stores the ranks for all possible k - 1 positions

(2) This costs us in total $O(\sqrt{n}lognloglogn)$ space, but we can compute rank in constant time

---

Efficient `rank` (Example)

first-level blocks: $\ell = 16$ second-level blocks: k = 4

	1								10								20		
B =	0	1	0	1	1	0	1	0	1	1	0	1	0	1	1	0	0	0	1	1	0	1
first =	0																9					
second =	0				2				4				7				0				2	

third =

	0 1 2
000	0 0 0
001	0 0 1
010	0 1 1
011	0 1 2
100	1 1 1
101	1 1 2
110	1 2 2
111	1 2 3

Restart counting in the next block.

$\lfloor \frac{19}{4} \rfloor \times 4 = 16$

$\lceil \frac{19}{4} \rceil \times 4 = 20$

$[17 \cdots 19]$

B[17] = 0

B[18] = 0

B[19] = 1

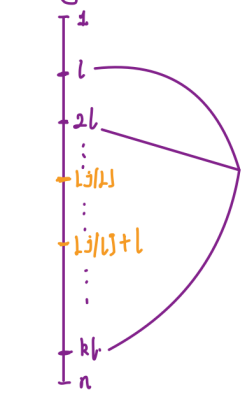
Example:

`rank`(B, 19) = first[19/16]
+ second[19/4]
+ third[001][(19 mod 4) - 1]

`rank`(B, 19) = first[1]
+ second[4]
+ third[001][2]

`rank`(B, 19) = 9 + 0 + 1 = 10

Binary Vector B.



Q: How to calculate count[B, j]

A: calculate ⌊j/l⌋ → first[⌊j/l⌋] = count[B, ⌊j/l⌋)

→ How many character c are there before ⌊j/l⌋

→ Then we walk through block [⌊j/l⌋, ⌊j/l⌋+l).

→ count(B[⌊j/l⌋*l+1 .. j) → count number of c.

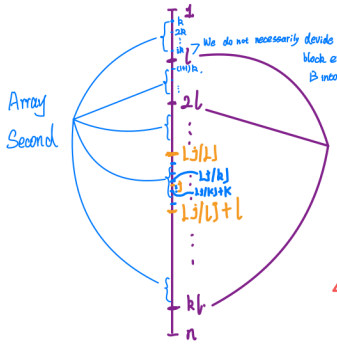
→ count(B, j) = first(⌊j/l⌋) + count(B[⌊j/l⌋*l+1 .. j])

→ Time: O(l), for $l=\lceil \log(n)/2 \rceil)^2$, we have $O(\log(n)^2)$

Space: $O(\frac{n}{l} \cdot \log(n)) \to O(\frac{n}{\log(n)^2} \cdot \log(n)) = O(n/\log n)$

For every block in first further devide them into seconds.

Binary Vector B.



Q: How to calculate count(B, j)?

A: count(B, j) = first(⌊j/l⌋) + count(B[⌊j/l⌋*l+1 .. j])

Q: How to calculate count(B[⌊j/l⌋*l+1 .. j])?

A: Start walking from max{ik ≤ j, i∈N}, Walk along B until j,

count(B[⌊j/l⌋*l+1 .. j]) = second(⌊j/k⌋) + count(⌊j/k⌋·k+1 .. j)

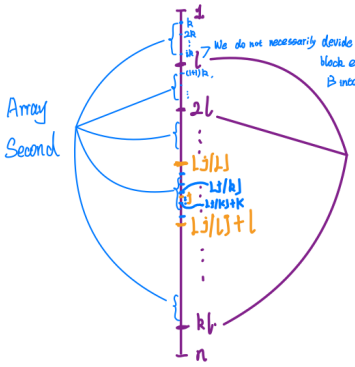
⚠ Instead of start walking at ⌊j/l⌋·l, we start walk ⌊j/k⌋·k and walk for at most k, which takes O(k). for k=⌈log n/2⌉, we have O(log n) time.

⚠: This costs additional O(n/k · log l) bits

$= O(n/\log n \log(\log n)^2)$

$= O(\frac{n \cdot \log\log n}{\log n})$

Binary Vector B.



For a length (k-1) binary block, there are 2^{k-1} possibilities.

	1	2	3	...	k-1
P_1	0	0	0	...	0
P_2	0	0	0	...	1
...	...	...	...	...	...
$P_{2^{k-1}}$	1	1	1	...	1

For every possibility P_i, we store the count/rank of the interested character. which takes

$O(k \cdot \log k) = O(\log n \cdot \log\log n)$ space.

AND, there're $O(\sqrt{n})$ possibilities,

So, table third takes

$O(\sqrt{n} \cdot \log n \cdot \log\log n)$ space.

But now we can compute

count(B, j) = first(⌊j/l⌋) + second(⌊j/k⌋) + count(⌊j/k⌋·k+1 .. j)

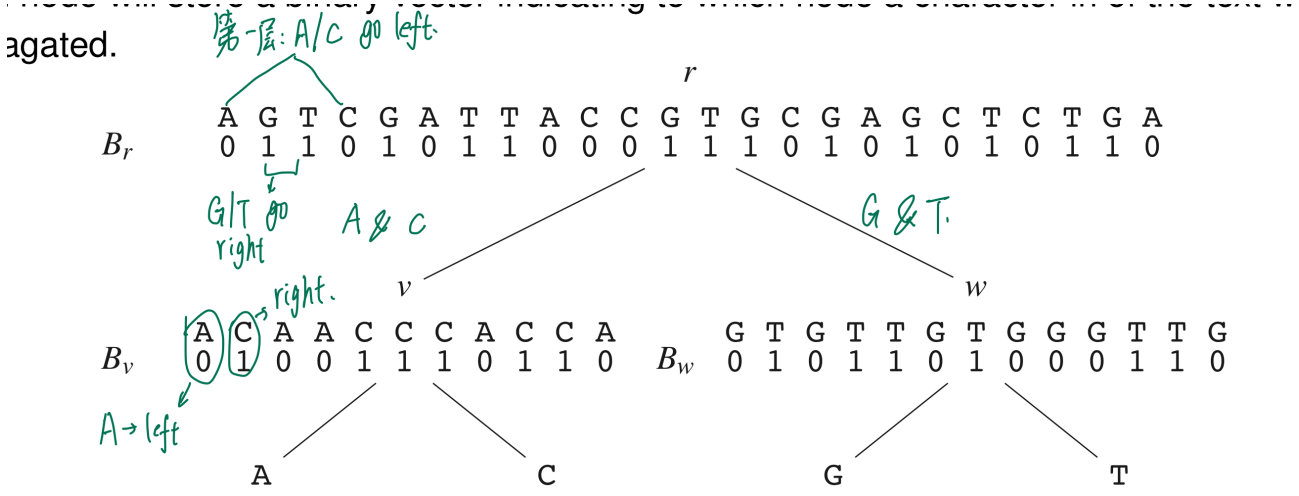
In linear time.

G. rank on integer vectors - wavelet tree

1. We can generalise the concept for rank and select from binary vectors to integer/character vector
2. A simple representation could be to encode each character in the alphabet with a binary vector and answer rank queries in constant time using $\sigma o(n)$ space.

	A	G	T	C	G	A	T	T	A	C	C	G	T	G	C	G	A	G	C	T	C	T	G	A
B_A	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
B_C	0	0	0	1	0	0	0	0	0	1	1	0	0	0	1	0	0	0	1	0	1	0	0	0
B_G	0	1	0	0	1	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0	0	1	0
B_T	0	0	1	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0

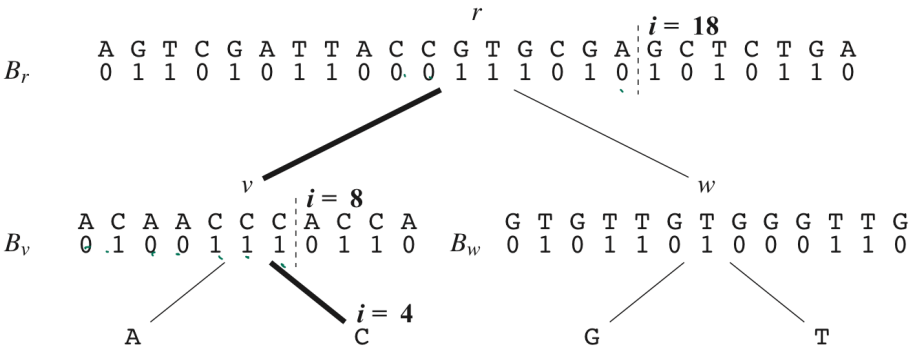
3. We can represent the data in less space providing the same query performance using a **wavelet tree**
4. **Wavelet trees**
   a) We will construct a balanced binary tree that shall efficiently split the alphabet at each level. Each node will store a binary vector indicating to which node a character in of the text will be propagated.



   b) This takes $O(nlog\sigma)$ space and let up perform $rank$ and $select$ in $O(log\sigma)$ time.

c) Example of using wavelet tree to calculate rank and select

Wavelet Tree - Example (`rank`)



Example $\texttt{rank}_C(T, 18)$:

C is left of the root ⟶ $\texttt{rank}_0(B_r, 18) = 8$

C is right of node v ⟶ $\texttt{rank}_1(B_v, 8) = 4$

Select 反问

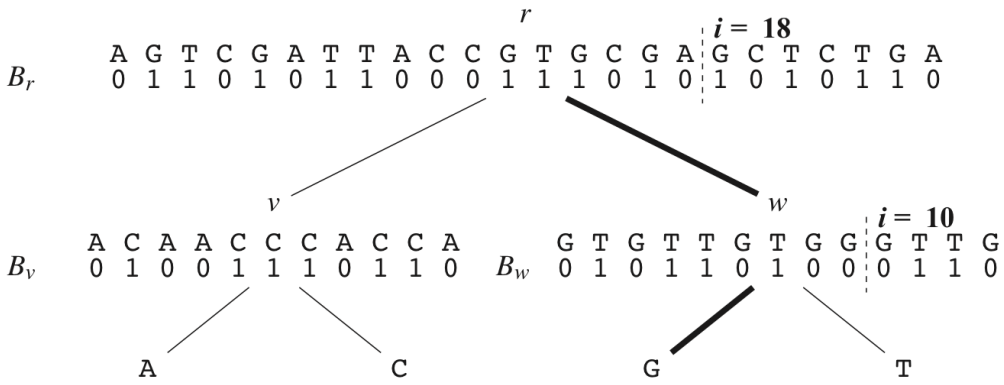
$\text{select}_C(T, 4)$

→ right on B_r: $\text{select}_1(B_r, 4) = 7$

left on B_r: $\text{select}_0(B_r, 7) = 15$

adapted from [Mäkinen et al., 201

Wavelet Tree - Example (`access`)



Example $\texttt{access}(T, 18)$:

$\texttt{rank}_{B_r[18]}(T, 18) = 10$ ⟶ $B_r[18] = 1$ ⟶ go right

⟶ $B_w[10] = 0$ ⟶ go left

adapted from [M

DINFK

AlgoPop – Dr. Andre

5. Operations on wavelet trees
   a) The operations rank, select, and access can be performed in $O(log\sigma)$ using a wavelet tree.
   b) In addition, the wavelet tree allows for two-dimensional range queries. The following operations can be carried out in $O(log\sigma)$ time.

(1) $range\_count(T, i, j, l, r) = |\{k \mid i \le k \le j, l \le t_k \le r\}|$: count the number of positions $\in [i, j]$ that contains character t_k within range $[l, r]$

(2) $freq_c(T, i, j) = rank_c(T, j) - rank_c(T, i-1)$: How often do we observe a certain character c within range $[i, j]$

(3) $isRangeUnary(T, i, j) = \begin{cases} true & \text{if } freq_{access(T,i)}(T, i, j) = j - i + 1 \\ false & \text{else} \end{cases}$

: measures if we only see character T_i at range $[i, j]$

H. Bi-directional BWT

1. Motivation:
   a) With the regular BWT, we can extend a pattern in only one direction. For many matching tasks, it is essential to efficiently extend a match in both forward and backward direction.
   b) Bi-derectional BWT : the key idea is that the search intervals in a forward and backward BWT of the same text are kept synchronised during extension / search
2. Preliminaries
   a) Given a text $T = t_1 t_2 \dots t_n$ with $t_i \in [1..\sigma]$ as well as its reversal $\overleftarrow{T} = t_n t_{n-1} .. t_1$, we can generate two BWT indexes:
      (1) $BWT_{T\#}$ built on $T\#$
      (2) $BWT_{\overleftarrow{T}\#}$ built on $\overleftarrow{T}\#$
   b) Let $\mathbb{I}(W, T)$ describes a function returning the interval in $BWT_{T\#}$ of all suffixes in T prefixed by W. Analog, let $\mathbb{I}(\overleftarrow{W}, \overleftarrow{T})$ return the interval of all suffixes in $\overleftarrow{T}$ prefixed by $\overleftarrow{W}$.
      (1) Then this leads to:
         (a) $\mathbb{I}(W, T)$ contains all positions i that are start positions of W in T
         (b) $\mathbb{I}(\overleftarrow{W}, \overleftarrow{T})$ contains all positions i such that $n - i + 1$ marks all end positions of W in T
   c) Now we can give the formal definition of bi-derectional BWT
3. Definition: Bi-directional BWT
   a) For a give nstring $T \in [1..\sigma]^n$, the bi-directional BWT is a data structure that supports the following operations:
      (1) $isLeftMaximal(i, j) = 1$ if $BWT_{T\#}[i..j]$ contains > 1 distinct characters
      (2) $isRightMaximal(i, j) = 1$ if $BWT_{\overleftarrow{T}\#}[i..j]$ contains > 1 distinct characters
      (3) $enumerateLeft(i, j) =$ all distinct characters in $BWT_{T\#}[i..j]$ in lex order
      (4) $enumerateRight(i, j) =$ all distinct characters in $BWT_{\overleftarrow{T}\#}[i..j]$ in lex order
      (5) $extendLeft(c, \mathbb{I}(W, T), \mathbb{I}(\overleftarrow{W}, \overleftarrow{T})) = (\mathbb{I}(cW, T), \mathbb{I}(\overleftarrow{Wc}, \overleftarrow{T})), c \in [0..\sigma]$
      (6) $extendRight(c, \mathbb{I}(W, T), \mathbb{I}(\overleftarrow{W}, \overleftarrow{T})) = (\mathbb{I}(Wc, T), \mathbb{I}(\overleftarrow{cW}, \overleftarrow{T})), c \in [0..\sigma]$
   b) More on $extendLeft$
      (1) The task of $extendLeft(c, \mathbb{I}(W, T), \mathbb{I}(\overleftarrow{W}, \overleftarrow{T}))$ is to update the interval of word math W to a new interval matching cW (or $\overleftarrow{Wc}$, respectively.)

(2) Beginning with the existing intervals, we would like to update:

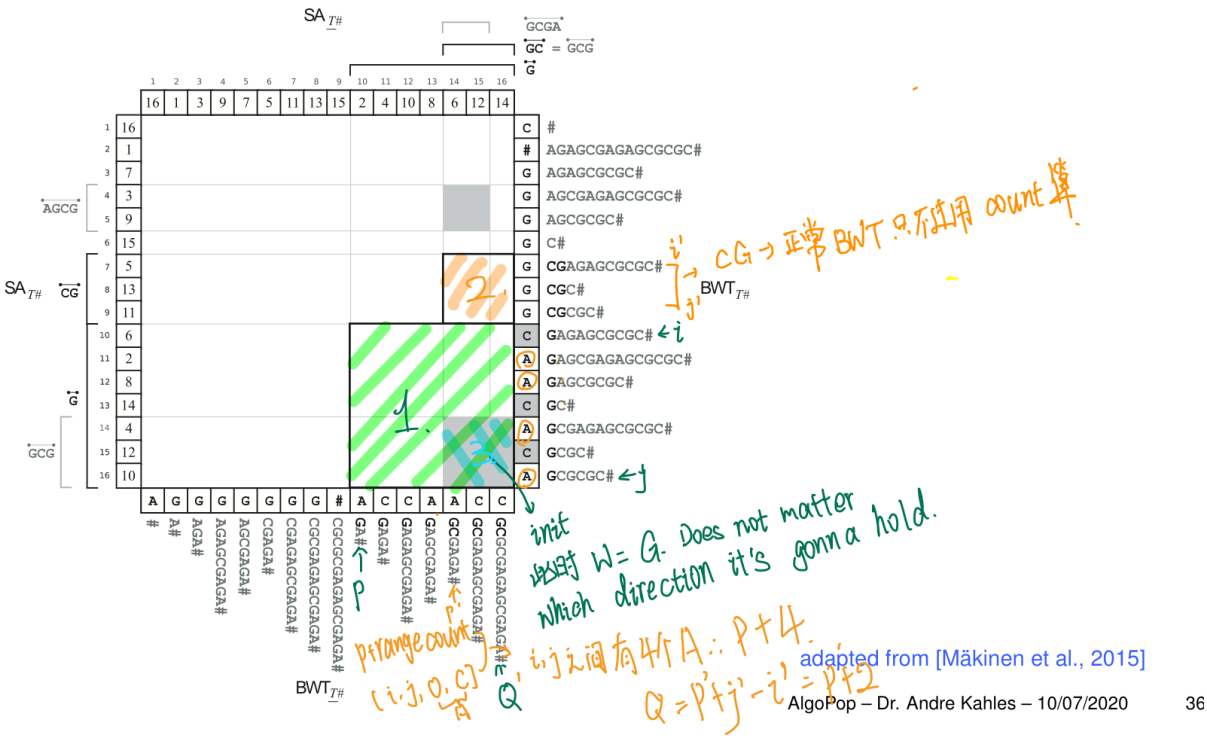
$$\mathbb{I}(W, T) = [i\,..\,j] \quad \Rightarrow \quad \mathbb{I}(cW, T) = [i'\,..\,j']$$
$$\mathbb{I}(\overleftarrow{W}, \overleftarrow{T}) = [p\,..\,q] \quad \Rightarrow \mathbb{I}(\overleftarrow{Wc}, \overleftarrow{T}) = [p'\,..\,q']$$

(3) We know that we can compute [i' ..j'] in $O(log\sigma)$ time by representing $BWT_{T\#}$ as a wavelet tree and utilising $rank_c$

(4) Since [p..q] contains all suffixes in $\overleftarrow{T}\#$ prefixed with $\overleftarrow{W}$, the interval $[p'\,..\,q']$ representing $\overleftarrow{Wc}$ needs to be contained in [p..q] and we compute:

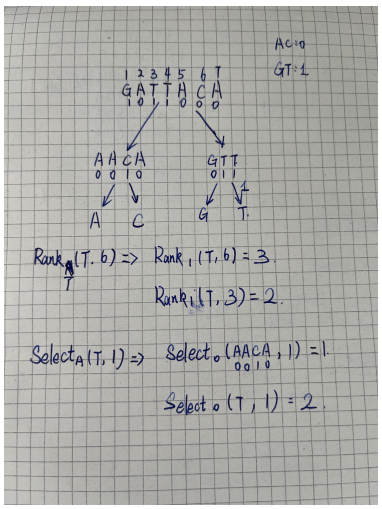
$$p' = p + range\_count(BWT_{T\#}, i, j, 0, c-1)$$
$$q' = p' + j' - i'$$

4. Example of Bi-directional BWT P = AGCG



IV. Example exam Questions

A. Given the string GATTACA, can you draw a schematic representation of the string in a wavelet tree?

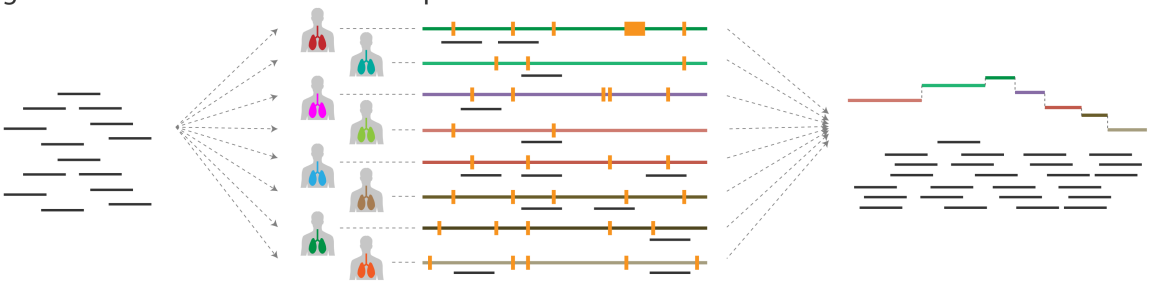


V. Summary for Burrows-Wheeler transform
   A. So far we have discussed classical BWT variants that enable efficient search within a string in one or two (bi-directional BWT) directions
   B. Using wavelet-trees to represent the BWT, we can achieve $O(log\sigma)$ time for any $rank()$ and $select()$ operation
   C. Based on this result, we can find all matches of a given pattern P in $O(|P|log\sigma)$ time

Lecture 5 - Variants of BWT

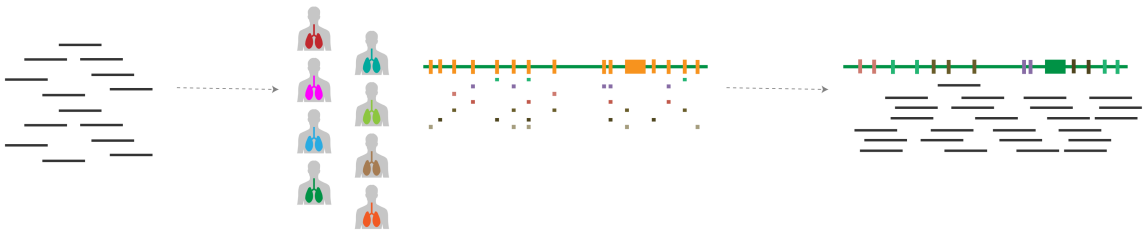
I. General motivation
   A. Complex genomic references
      1. Alignment to individual reference sequences
         a) Instead of using a single reference genome, we store the each individual genome from the population as a collection of reference genomes
            (1) get rid of the reference bias
         b) When a new individual got sequenced, we align the reads to the collection of individual reference genomes
            (1) for each read in the new individual, we try to find the best alignment within the whole collection of reference individual genomes.
         c) Genotype the new individual based on the optical alignment
         d) Problem : Very Expensive

Alignment to individual reference sequences

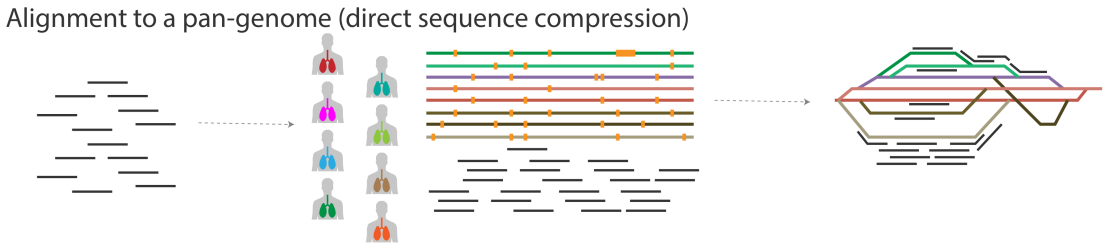


      2. Alignment to single reference + variation (reference based compression)
         a) Since most of the individuals in the population share most of their genomes, we can store the collection of individual reference genomes as one single reference + variations that are distinct to each individual

Alignment to single reference + variation (reference based compression)



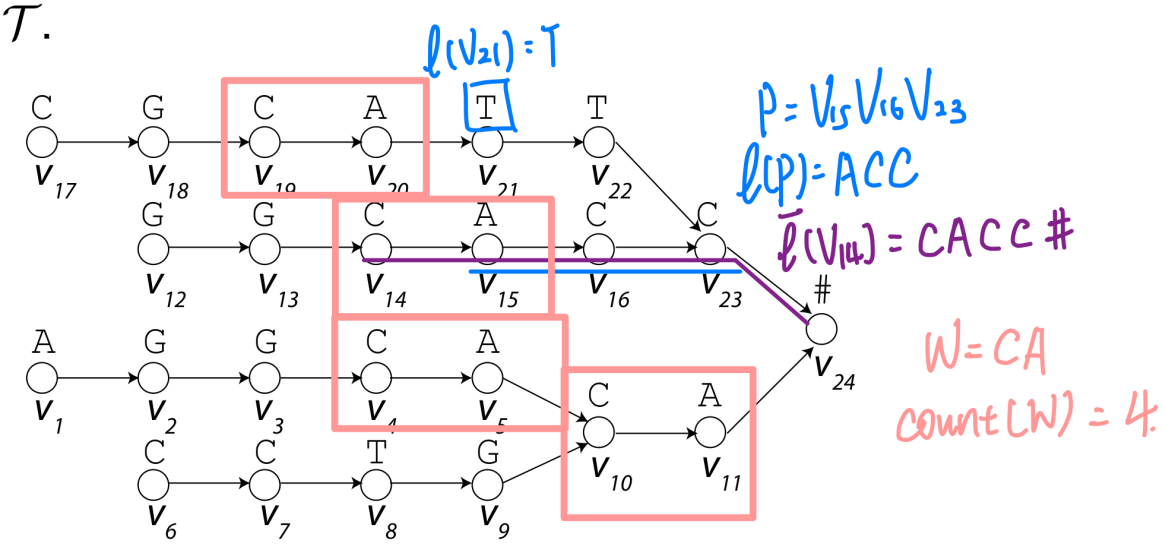
3. Alignment to a pan-genome (direct sequence compression)
   a) Compare the individual reference genomes directly and get a graph structure that would represent the comparison result
   b) Align the new individual genome to the graph



II. BWT on trees (principle, complexities, access)

A. Given a rooted, labeled tree $T = (V, E, \Sigma)$ with V being the node, E being the edges, Σ being the alphabet, and directed edges (u, v) from children to parents with each node labeled $\ell(v_i) \in \Sigma$, we define a path $P = v_i, v_{i+1}, \ldots, v_k$, such that $(v_i, v_{i+1}) \in E$ and a **path label** $\ell(P) = \ell(v_i) \cdot \ell(v_{i+1}) \cdots \ell(v_k)$. We further define the **extended label** $\bar{\ell}(v_i)$ as $\ell(v_i, v_{i+1}, \ldots, v_r)$ and assume that a **total order** $u \prec^* v$ between all children u, v of any given node, implying a total order on T

on $\mathcal{T}$.



B. Important Observations

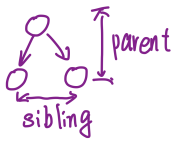
1. If tree T were a chain, $\bar{\ell}(v_i)$ would be a suffix of T
2. A string W can be the label of multiple paths, we use $count(W)$ to represent this number
3. A string W is the prefix of the extended label $\bar{\ell}(v_i)$ of $count(W)$ many nodes v_i

C. Definition of auxiliary data structures

1. For each node $v \in V$ we define

a) $last(v) = \begin{cases} 1, & \text{iff v is greater than any of its siblings} \\ 0, & \text{otherwise} \end{cases}$

b) $internal(v) = \begin{cases} 1, & \text{iff v is not a leaf} \\ 0, & \text{otherwise} \end{cases}$



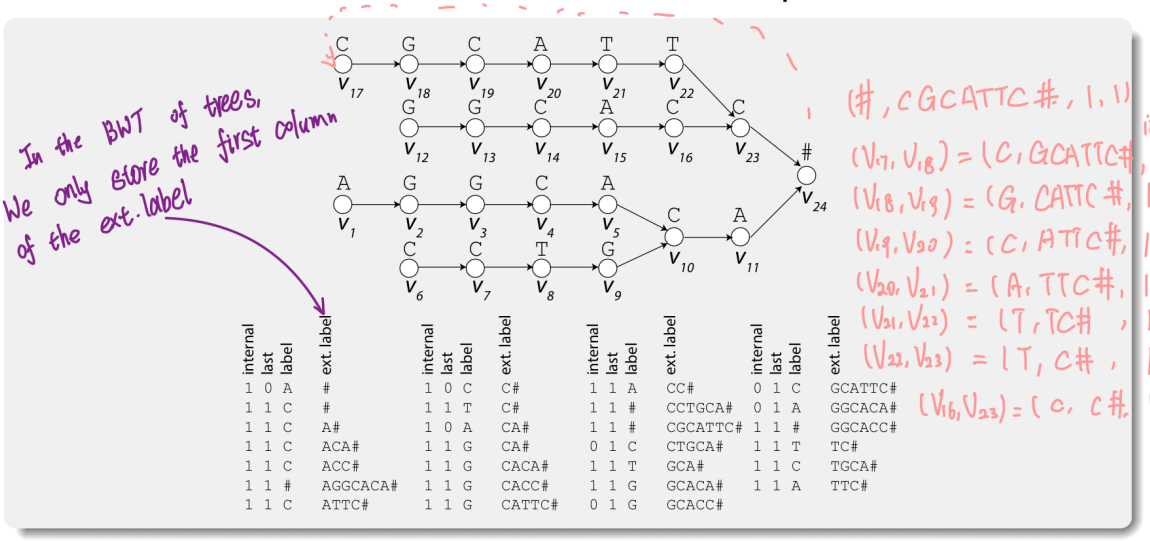
2. We now collect for all $(u, v) \in E$ (Here u is the child and v is the parent)in the order implied through $\prec^*$ a list L of 4-tuples:

$$L[i] = \begin{cases} (\ell(u), \bar{\ell}(v), last(u), internal(u), & \text{if v is internal} \\ (\#, \bar{\ell}(v), 1, 1), & \text{otherwise} \end{cases}$$

and apply a stable sort on L over the second element (the extended labels) of the tuples

3. Example of Burrows Wheeler Transform on Trees.

Burrows Wheeler Transform on Trees - Example



D. Definition of BWT on trees:

1. Burrows Wheeler index for trees
2. For a give labelled tree T, its **Burrows-Wheeler index** BWT_T consists of a data structure containing
   a) the array of characters labels
   b) the binary vector last
   c) the binary vector internal
   d) the first column of extended label encoded as offset vector C

internal	last	label	ext. label
1	0	A	#
1	1	C	#
1	1	C	A#
1	1	C	ACA#
1	1	C	ACC#
1	1	#	AGGCACA#
1	1	C	ATTC#

internal	last	label	ext. label
1	0	C	C#
1	1	T	C#
1	0	A	CA#
1	1	G	CA#
1	1	G	CACA#
1	1	G	CACC#
1	1	G	CATTC#

internal	last	label	ext. label
1	1	A	CC#
1	1	#	CCTGCA#
1	1	#	CGCATTC#
0	1	C	CTGCA#
1	1	T	GCA#
1	1	G	GCACA#
0	1	G	GCACC#

internal	last	label	ext. label
0	1	C	GCATTC#
0	1	A	GGCACA#
1	1	#	GGCACC#
1	1	T	TC#
1	1	C	TGCA#
1	1	A	TTC#

E. Traversal on BWT on trees

1. Observations:
   a) all node v sharing a prefix of their $\bar{\ell}(v)$ form a contiguous interval in BWT_T
   b) the in-degree of node v is directly encoded in $last$ and can be retrieved in constant time
      (1) Perform a $select$ on binary array last
      (2) For the example above, if we want to know the in-degree of CA#, we can perform a binary array starting at block CA#, which is 01, we perform a select like so $select_1(last_{CA\#},1) = 2$, and 2 is the in-degree of CA#
      (3) This can be done in constant time
   c) the k-th occurrence of c in labels corresponds to the k-th extended label beginning with c
      (1) In the k-th extended label beginning with c, we only count those extended label whose $last$ is 1
         (a) This is because we have more edges than nodes, we have to keep track using $last$
      (2) For the example above, the third occurrence in C above, corresponds to the third occurrences in ext.labels (neglecting C# and CA#, since their $last$ is 0.

2. Tree BWT Traversal - top-down
   a) Definition : Top-down Traversal
      (1) Given the range of node v in BWT_T as $\overleftrightarrow{v} = [p\,..\,q]$, we can compute the range of its k-th child with label c as :

$$child(\overleftrightarrow{v}, c, k) = [select_1(last, LF(p-1,c) + k - 1) + 1..$$
$$select_1(last, LF(p-1,c) + k)]$$

With $$LF(i,c) = C[c] + rank_c(labels, i)$$

   b) Definition : Left-extension of suffix W
      (1) Given a range $\overleftrightarrow{W} = [p\,..\,q]$ of nodes that all have string W as prefix for their extended label, we can compute the range of the **left extension** cW as:

$$extendLeft(\overleftrightarrow{W}, c) = [select_1(last, LF(p-1,c)) + 1..select_1(last, LF(1,c))]$$

internal	last	label	ext. label
1	0	A	#
1	1	C	#
1	1	C	A#
1	1	C	ACA#
1	1	C	ACC#
1	1	#	AGGCACA#
1	1	C	ATTC#

internal	last	label	ext. label
1	0	C	C#
1	1	T	C#
1	0	A	CA#
1	**1**	**G**	**CA#**
1	1	G	CACA#
1	1	G	CACC#
1	1	G	CATTC#

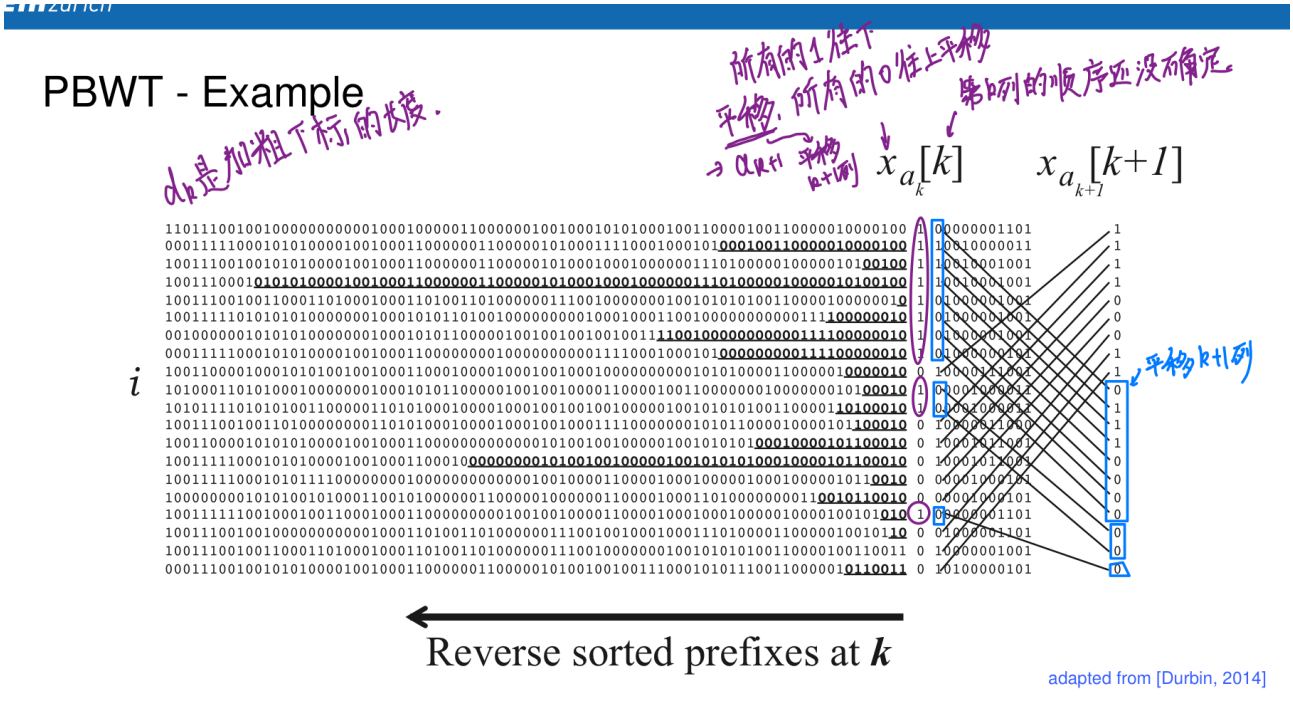
internal	last	label	ext. label
1	1	A	CC#
1	1	#	CCTGCA#
1	1	#	CGCATTC#
0	1	C	CTGCA#
1	**1**	**T**	**GCA#**
1	1	G	GCACA#
0	1	G	GCACC#

internal	last	label	ext. label
0	1	C	GCATTC#
0	1	A	GGCACA#
1	1	#	GGCACC#
1	1	T	TC#
1	1	C	TGCA#
1	1	A	TTC#

Count	
#	0
A	1
C	6
G	15
T	21

F. Pattern matching
   1. The top-down traversal is a generalisation of the backward search in a BWT on a string, Thus, we can search for a pattern P in $O(|P|log\sigma)$
   2. Analog to the top-down traversal, we can also implement bottom-up traversal with a complexity of $O(log\sigma)$ time
   3. With this, the tree is fully traversable in both directions, each step taking $O(log\sigma)$ time.

III. Positional Burrows Wheeler Transform (PBWT)
   A. Motivation: The idea of this structure was suggested as a storage and matching data structure for binary haplotype data.
   B. Idea: Instead of storing a matrix of M column-matched haplotype sequences of length N, store a permutation of the columns such that operations such as finding maximal matches between haplotypes can be carried out in $O(NM)$
   C. Key insight is to order each column k differently, such that prefixes up to position k are sorted inverse lexicographically
   D. Formal Definition of PBWT
      1. We are given a set X of different haplotypes x_i, all of length n, with $i \in [0..m-1]$. We encode each haplotype as a binary string, so $x_i[k] \in \{0,1\}$
      2. For a given k, we can now define an array a_k that holds the sort index over all haplotypes sorted by their prefixes $x_i[0..k-1]$ in reverse lexicographic order
      3. We also define the array of longest common suffixes $d_k[i]$ such that
$$d_k[i] = \min_{j \in [0..k-1]} \{j \,|\, x_{a_k[i]}[j..k-1] = x_{a_k[i-1]}[j..k-1]\}$$
      4. It can then be shown that the start of any maximal match ending at k between haplotypes $x_{a_k[i]}$ and $x_{a_k[j]}(i<j)$ is given as $\max_{i<p\le j} d_k[p]$
   E. Example



F. Complexity
1. The arrays a_k and d_k only depend on the sorting order of previous column $k-1$
2. we can sweep through all columns once to compute all sort orders in O(mn) time
3. We will now see how this can be exploited to also perform haplotype matching in O(mn) time

G. Haplotype Matching
1. Goal: We would like to report all matches within X of length greater than some threshold L ending at position k.
2. We can achieve this utilising arrays a_k and d_k at each column k. Observation: A match between sequences i and j in sort order of column k ends at k, if
$$x_{a_k[i]}[k] \neq x_{a_k[j]}[k]$$
Further, as all matching suffixes have to be adjacent (in sort order of column k) to get only matches of length greater than L we can require:
$$d_k[m] \leq k - L \quad \forall i < m \leq j$$
All matches ending at k only depend on d_k and a_k, and can be computed in the same sweep used to compute d_k and a_k, taking O(mn) time
3. This logic can be extend to also generate all set-maximal matches of haplotypes x_i in X in linear time
   a) Set-maximal match
      (1) A match of haplotype x_i to another haplotype $x_j (i \neq j)$ from k_1 to k_2 is **set-maximal** if it can not be extended to left or right and no longer match of x_i to another x_k $(i \neq k, k \neq j)$ including $[k1..k2]$ exists
   b) We can compute such maximal matches for each haplotype x_i in the same sweep over X and thus can report them in $O(nm)$ time (as we only report matches ending at column k).
   c) For each $x_{a_k[i]}$ search for a range [p..q] such that:
$$d_k[j] \leq min(d_k[i], d_k[i+1]) \quad \forall p < j \leq q$$

H. Compression
1. Instead of storing X and the arrays a_k and d_k, one can also directly store the permuted version of each column x_{a_k}. We call the resulting matrix Y, the **positional burrows wheeler transform** of X
2. Similar to BWT on linear strings, the compression performance of PBWT originate from correlation structure in the input. For haplotypes this correlation structure can be explained with the linkage disequilibrium between neighbouring variant sites.
3. With Y alone, we can efficiently compute all long matches starting at any column of X. However, if we want to project back into the order of X, we need to keep sparse versions of a_k and d_k too.

IV. Graph Positional Burrows Wheeler Transform (gPBWT)

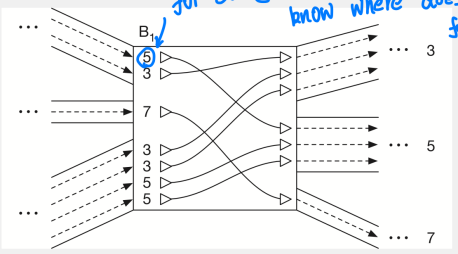
A. The concept of PBWT has been extended to also work for indexing threads (a generalised form of haplotypes) in a genome graph representation.

B. key idea: Instead of representing allelic state at a certain position, encode the index of the next child visited by a thread

C. graph positional burrows wheeler transform is used to embed to the genome variation on page 24 with the information of which genome has actually been observed in my dataset. This kind of information can be very useful when it comes to alignment to the graph.
   1. Which two genes are actually observed in one sample and at which frequency

child visited by a thread.

Example



- indices of next nodes in each thread are stored in a vector B_k for node k
- max two vectors per node (one per direction)

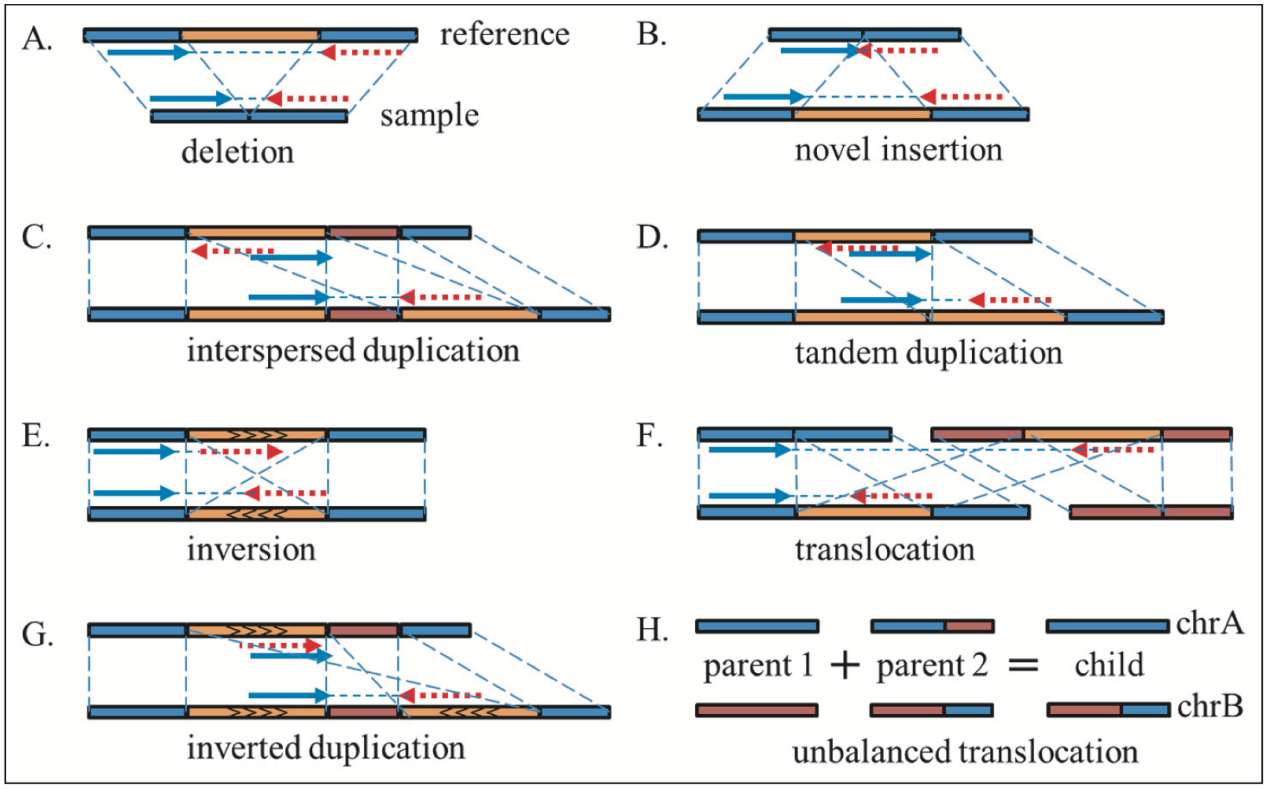
V. gPBWT-More formal
   A. Given a graph $G = (V, E)$, each node $v \in V$ has two sides and each edge $e \in E$ is a pair set of sides. A thread t is a non-empty sequence of sides, such that for all $0 \leq i \leq N$, sides t_{2i} and t_{2i+1} are opposites of each other and $\exists e \in E$ connecting t_{2i} and t_{2i+1}
   B. We now want to store a set of threads T embedded into the graph. Every side t_{2i} in a thread represents a visit to a node.
   C. For each side, we locally sort all threads visiting inverse lexicographically by the suffix of the sequence of previous visited sides. Following this order, we collect the indices of the next sides visited by each of the threads. We call this vector B_s for a given side s
   D. Further, given an oriented edge (s, s'), we define a function c((s, s')) that returns the smallest index in $B_{s'}$ of a visit at s' traversing {s.s'}
      1. function c is direction part
   E. The combination of function c() **and all arrays B[] form the gPBWT.**

VI. Example Questions
   A. Why would we want to index anything else than strings?
      1. Aligning to one single reference genome may cause the problem of reference bias. So we want to align to a population of individual reference genomes. However, it is too expensive to actually align a new sequenced genome to all those populations of reference genomes.
      2. So, we need to index the population of individual genomes, for that reason, we can index the genome graph and align the newly sequenced genome to the indexed genome graph.
   B. Given a trie over alphabet Σ as discussed in the lecture, what is the equivalent to a suffix in a strong over Σ?
      1. Extended path labels

Lecture 6 - Alignment-based genome comparison

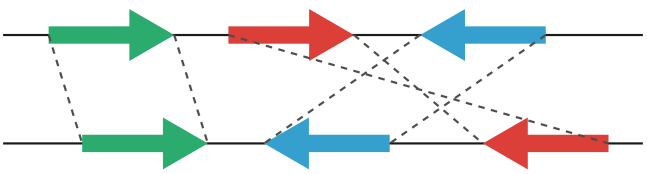
I. Motivation
   A. The research field of comparative genomics addresses many questions that are based on comparison of genomes:
      1. Uncover evolutionary relationships based on whole genome sequences
      2. Identify conceived sequence elements and derive functional roles
      3. Understand mechanisms and principles of genome evolution
      4. Learn about the regulatory landscape and organisational structure of genomes
      5. Identify structural variation

II. Classes of structural variation



III. Challenges
   A. The comparison of multiple whole genome sequences is a non-trivial task. Classical alignment-based approaches struggle because:
      1. The size and number of sequences to be compared is too large
      2. certain re-arrangements can not be expressed through alignment in a dynamic programming matrix.(e.g. translocations, inversions, …)
   B. Possible solutions:
      1. Use local alignments as anchors
      2. Base alignment on related sub-sequences (e.g. gene sets)
      3. Use alignment-free distance measures

IV. Alignment based methods
   A. Most alignment based methods chain together highly similar local segments into compatible paths

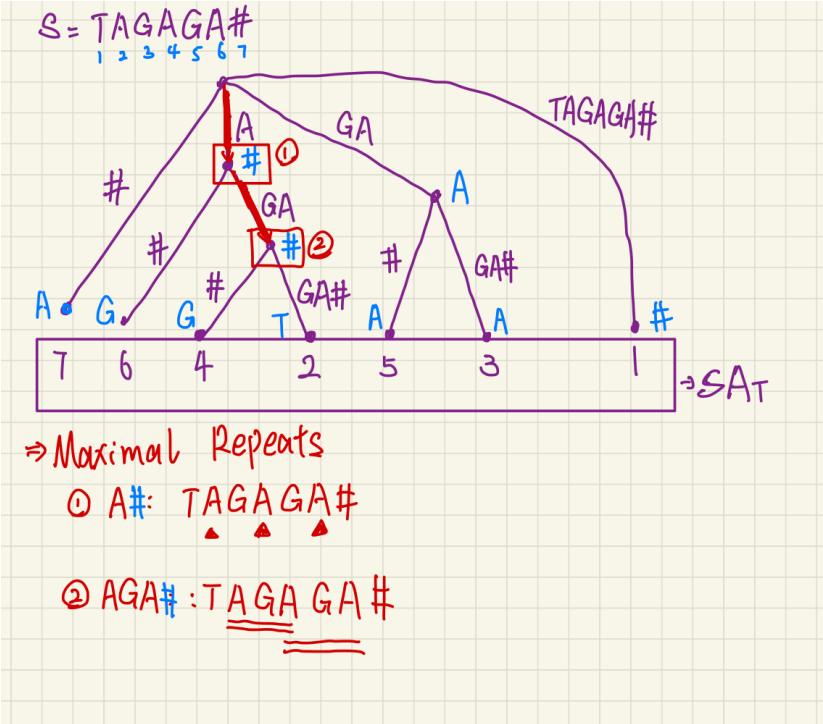


B. Match maximality
   1. To make matches identifiable, we need to introduce the concept of maximality
   2. **Right-maximal**: A match between two strings S and T is called right-maximal, if it can not be extended to the right, without loosing at least one of its occurrences. If the matching strings are $s_i s_{i+1} \dots s_{i+k-1}$ and $t_j t_{j+1} \dots t_{j+k-1}$ it follows that $s_{i+k} \neq t_{j+k}$
   3. **Left-maximal:** A match between two strings S and T is called left-maximal, if it can not be extended to the left without loosing at least one of its occurrences. If the matching strings are $s_i s_{i+1} \dots s_{i+k-1}$ and $t_j t_{j+1} \dots t_{j+k-1}$, it follows that $s_{i-1} \neq t_{j-1}$

C. Maximal repeats
   1. A **maximal repeat** of string $T = t_1 t_2 \dots t_n$ is a substring X that occurs more than once in T and that cannot be extended in either direction without losing one of its occurrences.
   2. We find all maximal repeats, by a linear time post-order traversal of suffix tree ST_T.
      a) We define for all leaves v the function $prev$ as $prev(v) = T[i-1]$ where $T[i\,..\,n]$ is the suffix corresponding to the leaf.($prev(r) = \#$ if the label of r equals T.) For all inner nodes u we define recursively:

$$prev(u) = \begin{cases} \# & \text{if } \exists \geq 2 \text{ leaves in sub-tree of } u \text{ with suffixes} \\ & T[i\,..\,n] \text{ and } T[j\,..\,n] s.t. T[i-1] \neq T[j-1] \\ c \in [1..\sigma] & c = T[i-1] \forall T[i\,..\,n] \text{ of leaves under } u \end{cases}$$

   3. All paths of inner nodes with label # are maximal repeats of T.
   4. Example of maximal repeats



D. Maximal Unique Match (MUM)

1. **Maximal Unique Match:** A maximal unique match(MUM) of two strings S and T is a substring X that occurs in both S and T exactly once and cannot be extended in either direction without losing the occurrence/
2. We can find the MUM of S and T efficiently, using the suffix tree ST_C built on the concatenation $C = S\$T\#$
3. Useful Observations:
   - **a)** MUMs are right-maximal substrings of C (as are the paths to inner nodes of a suffix tree)
   - **b)** Only internal nodes of ST_C with exactly 2 leaves as children can be MUMs.
4. Algorithm
   - **a)** Traverse ST_C and find all inner nodes with exactly two leaves as children whose corresponding suffixes $C[i\,..\,|\,C\,|\,]$ and $C[j\,..\,|\,C\,|\,]$are such that $i \leq |\,S\,|$ and $j > |\,S\,| + 1$
   - **b)** Check whether the match between such leaves is left maximal $S[i-1] \neq T[j-1]$
5. Time complexity
   - **a)** We can find all MUMs in $O(|\,S\,| + |\,T\,| + \sigma)$ time
   - **b)** This result can be generalised to sets of strings, using their concatenation, to a total time $O(N + \sigma)$, where N is the total length of all strings

E. Bidirectional BWT vs Suffix Trees

1. Given a text T, we can connect its bi-directional BWT ($BWT_{T\#}$ and $BWT_{\bar{T}\#}$) with the suffix tree ST_T
2. Useful Observation: The path from the root to an inner node of suffix tree ST_T corresponds to prefix shared by multiple suffixes in T and hence to a contiguous interval in bi-directional BWT
3. We can exploit this to find all inner nodes of ST_T (which is required to find the MUM and maximal repeat) using a linear time traversal through the bi-directional BWT. We can complete this in $O(n \log \sigma)$ time, given that the BWTs are implemented using wavelet trees.
4. Algorithm to find inner nodes of ST_T

   a) For a given text $T \in [1..\sigma]^n$:

   (1) Traverse through $BWT_{T\#}$ and $BWT_{\bar{T}\#}$ using the intervals $[i\,..\,j]$ and $[i'\,..\,j']$, respectively. Initialize $[i\,..\,j] = [i'\,..\,j'] = [1..n+1]$, corresponding to the root of ST_T

   (2) Push the tuple $((i,j),(i',j'),d) = ((1,n+1),(1,n+1),0)$ to a stack S, where d is the depth of ST_T

   (3) For each entry in S

   (a) report interval (i,j) as inner node

   (b) collect a list $\mathcal{J}$ that contains the left extensions of (i,j) for any $c \in BWT_{T\#}[i\,..\,j]$

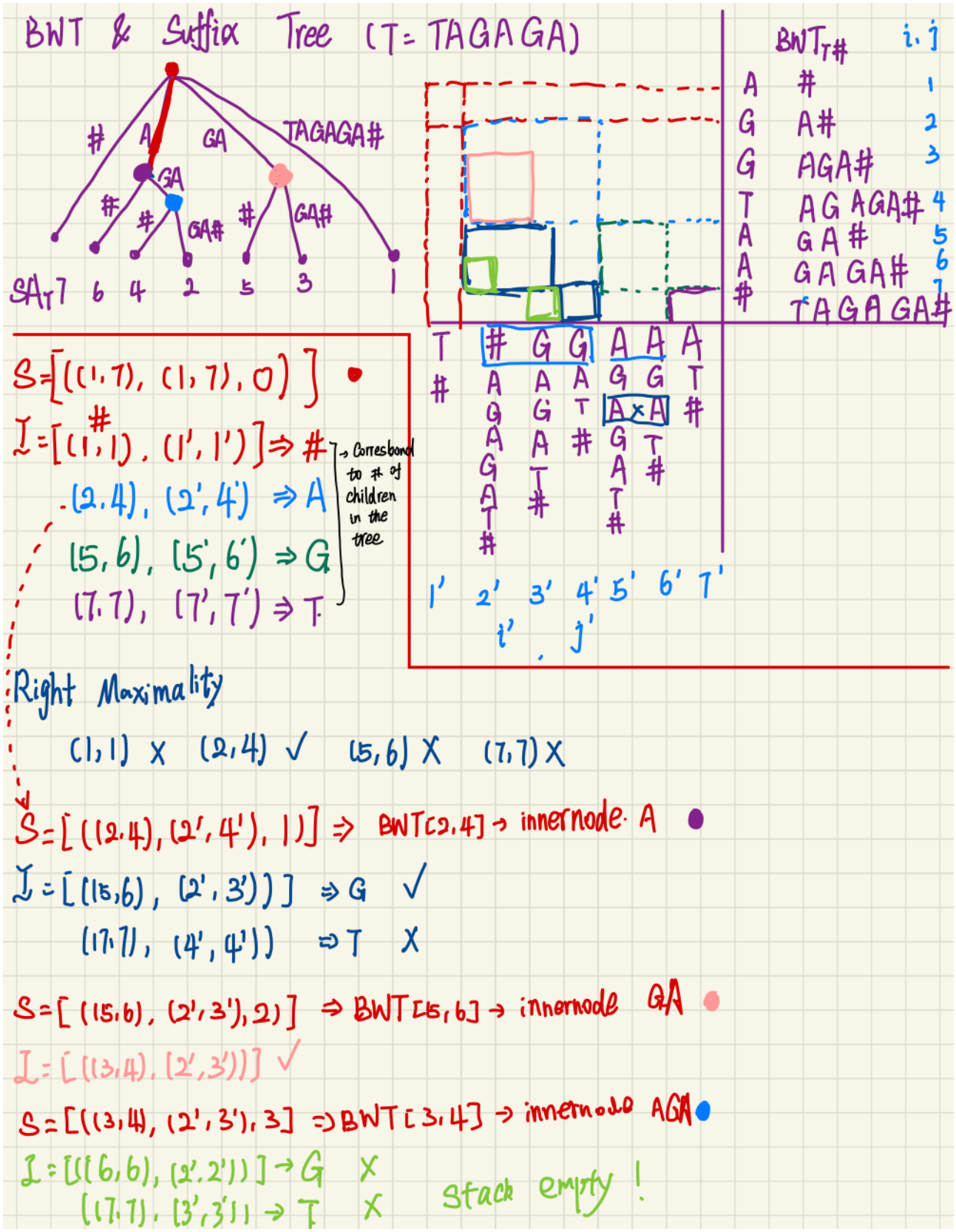
   (c) $\forall (i,j) \in \mathcal{J}$, check for right-maximality using $BWT_{\bar{T}\#}[i'\,..\,j']$

   (d) push all right-maximal elements of $\mathcal{J}$ as new elements to S, increase the depth d by 1

(e) iterate until S is empty

b) This algorithm is a lot like a breadth-first search of a given tree

5. Example



F. Maximal repeats for full genomes

1. The algorithms discussed so far do not scale well to full genomes. We utilise efficient data structures like the BWT to improve them.
2. Goal : Extend the simple suffix tree algorithm to utilise a bi-directional BWT instead.
3. Observation: inner nodes of a suffix tree correspond to suffixes that share a common prefix. These prefixes are already right-maximal.
4. Adaptation of our approach visiting all inner nodes of ST_T
   a) For reporting inner nodes, we already checked for right maximality
   b) we only need to also check for left-maximality before we output
5. Any path from the root to an inner node in the suffix tree ST_T that is also left-maximal is a maximal repeat of T!
6. We can do this in $O(n \log \sigma + occ)$ time

G. MUMs for whole genomes
1. A similar modification can be used for maximal unique matches between two given strings S and T. (Achieving a time of $O((|S| + |T|)\log \sigma)$)
2. Again, we use the concatenation $R = S\$T\#$ but will now build a bi-directional BWT instead of a suffix tree.
3. Observation: We are looking for maximal repeats in R with the restriction that one occurs in S and the other in T
4. We will make the following modifications to our algorithm:
   a) Introduce an indicator vector $\mathcal{I} \in [0,1]$ s.t. $\mathcal{I}[i] = 0$ iff the *i*-th suffix in lexicographic order in R is from S (and 1 otherwise)
   b) for each processed interval $(i\,.\,j)$ count the number of suffixes from S and T using $rank$ on $\mathcal{I}$ (e.g., $rank_o(\mathcal{I}, j) - rank_0(\mathcal{I}, i)$)
   c) only output, if interval (i, j) is left-maximal and $\mathcal{I}(i, j)$ contains exactly one 0 and one 1.

H. Maximal exact matches (MEM)
1. We can generalise the idea of MUM and relax the uniqueness condition. The general idea of an algorithm then follows closely the one for MUMs.
2. Algorithm(sketch):
   a) We will focus on the modification relative to the MUM algorithm.
   b) Instead of building a bi-directional BWT on the concatenation of S and T, we build one bi-directional BWT for each string
   c) We keep track of the corresponding interval pairs in $BWT_{S\#}$ and $BWT_{T\#}$
   d) We only need to check for left-maximality before output (and need some extra book-keeping for special cases)
3. We can complete this task in $O((|S| + |T|)\log \sigma + occ)$ time, where occ is the size of the output

I. Application to the whole genome alignment.
1. Different approaches use MUMs or MEMs as anchors that are bases for finding the complete alignment through chaining or local extension.
2. In addition to exact matches, also methods using approximate matches exist, which increases sensitivity of anchor finding in more distant genome sequences.
   a) Use of anchors off length k but allowing for up to c mismatches
   b) use of spaced anchors only requiring matches on a certain subset of positions.

V. Example Exam Questions
   A. What are the typical use cases for alignment-based genome comparison?
      1. Uncover the evolution history based on whole-genome alignment
      2. Identify conceived sequence elements and derive functional roles
      3. Understand mechanisms and principles of genome evolution
      4. Learn about the regulatory landscape and organisational structure of genomes
      5. Identify structural variation
   B. What is the difference between MUMs and MEMs?
      1. MUMs: Maximal Unique Matches. Given two strings S and T, MUMs are sequences that are in both S and T exactly once and can not be extended on both directions without losing its occurrences.
      2. MEMs: Maximal Exact Matches, it is related MUMs but relaxes the requirement that it should be unique.

Lecture 7: Alignment-free genome comparison

I. Use cases for alignment free methods
   A. Alignment-based vs alignment-free genome comparison
      1. Alignment based:
         a) allows for structural comparisons
         b) insights to mechanistic details
         c) access to exact underlying sequences
      2. Alignment-free
         a) both faster and more space efficient
         b) allow for scalable comparative genomics
         c) genome identification through fingerprinting
   B. Motivation of alignment-free genome comparison
      1. The complexity of genome alignments steadily grows with the number and size of the genomes involved. We are looking for measures to assess the similarity / distance between genomes without alignment
      2. Several strategies have been developed, which we will discuss further
         a) string kernels
         b) document fingerprinting and winnowing schemes
         c) hushing-based comparison
II. String-kernels (principle)
   A. Kernel functions
      1. The concept of *kernel functions* originates from pattern analysis in machine learning
      2. Given a set of vectors $S \subset \mathbb{R}^n$, where each element of S belongs to one of two possible classes. We consider the classification algorithm $\mathscr{A}$ that divides the two classes with a hyperplane in $\mathbb{R}^n$.
      3. Assume now that the two classes can not be linearly separated in $\mathbb{R}^n$.
      4. Idea: Create a non-linear map $\phi : \mathbb{R}^n \mapsto \mathbb{R}^m$ with $m > n$, s.t. the problem becomes linearly separable in $\mathbb{R}^m$. Then we can use $\mathscr{A}$ on set $s' = \{\phi(X) \,|\, X \in S\}$ to solve the problem in $\mathbb{R}^m$
      5. Assume that algorithm $\mathscr{A}$ only uses inner products $X \cdot Y$ between pairs $\{X, Y\} \subset S$
      6. Problem: Carrying out such computation in $\mathbb{R}^m$ could be very costly

7. Solution: Find a symmetric function $k()$, such that $\phi(X) \cdot \phi(Y) = k(X, Y)$.
8. The family of functions $k : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}$ for which above mapping exists are called **kernel functions** and they are defined through *Mercer's* condition
9. The symmetric positive semi-definite matrix containing $k(X, Y)$ for all $\{X, Y\} \subset S$ is called the gram matrix of S.

B. String kernels
1. Given two strings S and T, we convert them into *composition vectors* $S, T \subset \mathbb{R}^n$, $n > 0$, containing all substructures of S and T of a specific type.
2. Such substructure could for instance be all $k-$mess (substrings of length k)
3. There is a number of additional useful properties that kernels have that are further discussed in learning theory. Here, we will focus on the algorithmic aspects only

C. Similarity/distance measure
1. As a similarity, we will use a variety of cosine similarity, generally defining the kernel function k as follows(where W is an element of the set of substructures. e.g., k-mers):
$$k(S, T) = \frac{\sum_W S[W]T[W]}{\sqrt{(\sum_W S[W]^2}\sqrt{\sum_W T[W]^2}}$$
2. As a similarity measure $k(S, T) \in [-1,1]$, We can transform it into a distance:
$$d(S, T) = \frac{1 - k(S, T)}{2} \in [0,1]$$
3. We will use these similarity/distance measures for a variety of substructures W
4. Example of Cosine Similarity
   a) S = ACAC
   b) T = CACAC
   c) for 2-mer AC and 2-mer CA, we have
   d) $S = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \qquad T = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$
   e) Then we calculate the cosine similarity as $\frac{2 \times 2 + 1 \times 2}{\sqrt{2^2 + 1^2}\sqrt{2^2 + 2^2}}$

D. Computing string kernels*
1. We can compute k-mer/substring complexity and k-mer/substring kernels efficiently using suffix trees. Following we will discuss the k-mer complexity/ kernel as example
2. Building on the space efficient algorithms using bi-directional BWT, we will also follow the same strategy for these computations. We will begin with k-mer complexity
3. Observation: A k-mer of S is either the label of a node v in suffix tree ST_S or the prefix of an edge (u, v)
4. Algorithm(sketch):
   a) initialize k-mer complexity $C(S, k)$ to $|S| - k + 1$
      (1) maximum number of k-mers
   b) enumerate all internal nodes of ST_S setting $C(S, k) = C(S, k) + d(v)$:

$$d(v) = \begin{cases} 0 & \text{if } |\ell(v)| < k \\ 1 - c(v) & \text{else, where } c(v) \text{ is the number of children of v} \\ & \text{meaning this k-mer would appear number of children times} \end{cases}$$

III. Document Fingerprinting
   A. A general strategy for comparing two texts (or documents. or genomes) is to use a technique called **fingerprinting**
   B. The goal is to compute a short summary of either the whole document or substrings of the document. that well reflects the content.
   C. There is a wide range of applications:
      1. plagiarism detection
      2. duplicate finding in file systems
      3. approximate search in web graphs
   D. Hashing
      1. The **goal** of a hash function is to map values from a very large (infinite) input range into a small output range of a pre-defined fixed size
      2. To be practically useful, hash functions need to fulfil a set of conditions:
         a) deterministic: The same input needs to generate the same output
         b) uniform: The output range shall be evenly mapped with values
         c) (non-invertible): the original value can not be computed from hash (application dependent, e.g. cryptography)
      3. Examples for hash functions are:
         a) modulo operation
         b) iterated digit sum
      4. More complex hashes usually employ a combination of modulo-operations, bit-shifting and binary logical operations.
   E. Rolling hashes
      1. Assume, we would like to compute a hash function, using a k-mer as input. When computing hashes of all consecutive k-mers in a text, we can exploit the sequence information and improve on the trivial $O(k \cdot |T|)$ complexity
      2. Simple example algorithm:
         a) treat $k-$mer as k-digit number of some base b
         b) compute hashes as
            $H(t_i \,..\, t_{i+k-1}) = t_i \cdot b^{k-1} + t_{i+1} \cdot b^{k-2} + \ldots + t_{i+k-2} \cdot b + t_{i+k-1}$
         c) upon shifting compute the update as
            $H(t_{i+1} \,..\, t_{i+k}) = (H(t_i \,..\, t_{i+k-1} - t_i \cdot b^{k-1}) \cdot b + t_{i+k}$
         d) optimised for more uniform hashing
            $H'(t_{i+1} \,..\, t_{i+k}) = ((H'(t_i \,..\, t_{i+k-1}) - t_i \cdot b^k) + t_{i+k}) \cdot b0 \bmod p0 \bmod$
   F. 0 mod p fingerprinting
      1. A simple strategy for document fingerprinting is the 0 mod p strategy:
      2. Given a text $T = t_1 t_2 \,..\, t_n$, compute all k-mers $t_i \,..\, t_{i+k-1}$, $\forall 1 \leq i \leq n - k + 1$. From each k-mer compute a hash value $h_i = H(t_i \,..\, t_{i+k-1})$. where H is an appropriate hash function.
      3. The fingerprint of text T will consist of the subset of hash values h_i with
         $$h_i \bmod p = 0$$
      4. Observation: The gap between any two fingerprinted k-mers can be arbitrarily large (depending on structure of T)

5. We will now discuss **winnowing** which takes this into account

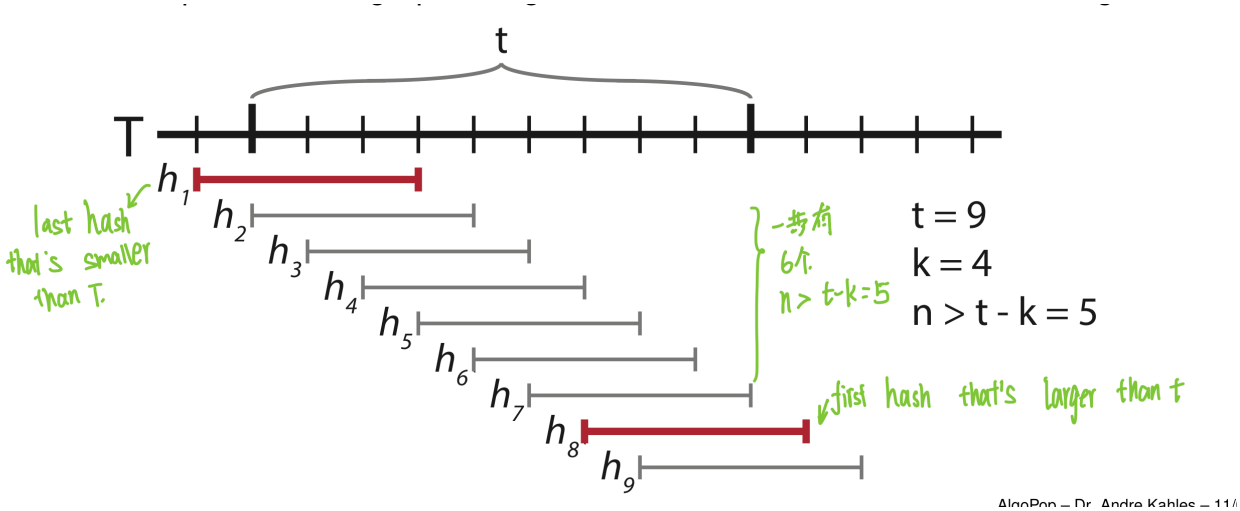
IV. Winnowing/minimizers

A. Motivation:

1. Goal: Given two strings S and T, detect all exact matches between the two strings such that:
   a) They have a length of at least k (noise threshold)
   b) matches of length at least t are guaranteed to be found
2. **Note**: Varying k shows the typical sensitivity-specificity trade-off.
3. Question: How can we meet the above guarantees while minimising the size of the fingerprints that need to be stored.

B. Idea

1. Observation: Given a sequence of consecutive hashes $h_i \ldots h_n$ if $n > t - k$, then at least one of the h_i must be part of the fingerprint to guarantee detection of all matches of length at least t



2. If non of the k-mers in $h_2 \ldots h_7$ are selected as fingerprint, then we would violate the goal above because there is no matches of at least length k in the length t area to be found.

C. Approach:

1. Use a window of size $w = t - k + 1$
2. Each position in T with $1 \leq i \leq n - w + 1$ defines a window of hashes $h_i \ldots h_i + w - 1$
3. selecting one hash form each window maintains the above guarantee.

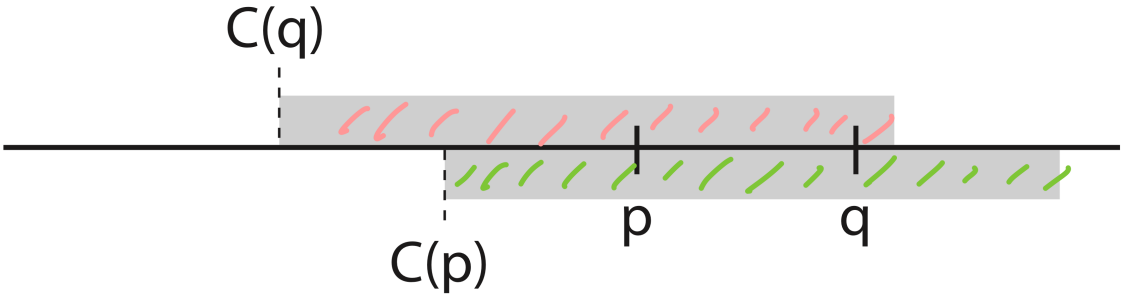
D. Definition : Winnowing

1. Given a sequence of hashes $h_1 \ldots h_n$ and a window of size w, the strategy to select for each window $h_i \ldots h_{i+w-1}$ the hash with minimal value as a fingerprint is called **winnowing**. If more than one hash has the same minimum value, the right-most one is selected

E. Expected density

1. Density: We define the **density** of a fingerprinting scheme as the fraction of fingerprints selected from the set of all computed hash values.

2. Consider a function C that maps the position of each selected fingerprint to the left-most window that selected it. We say that C charges the cost of fingerprint i to the window $C(i)$.
3. The charge function C is monotonically increasing: For any 2 fingerprints $p < q \rightarrow C(p) < C(q)$
   a) Proof by contradiction
   b) Assume for two chosen fingerprints p and q we have p < q and $C(p) \geq C(q)$. Since $p$is the chosen hash value from the green area, we know that p is the right-most smallest value. From the same sense, we know that q must be the right most smallest value in the pink area. This is contradictory to $p < q$, so $C(p) \leq C(q)$



4. Consider an indicator variable
$$x_i = \begin{cases} 1, & \text{iff window } W_i \text{ is charged} \\ 0, & \text{otherwise} \end{cases}$$
5. When p is the smallest hash in $t_{i-1} \ldots t_{i+w-1}$, it is chosen as fingerprint for both W_{i-1} and W_i. Leave 3 cases:
   a) $p = i - 1 \Rightarrow X_i = 1$ (p charged to W_{i-1}; choose some other pos. in W_i and charged to W_i)
   b) $p = i + w - 1 \Rightarrow X_i = 1$ (W_i is the left-most window using p)
   c) otherwise $\Rightarrow X_i = 0$ (W_i is not the left-most window using p)
6. Assuming uniformity over positions, the expectation over X is
$$\mathbb{E}(X) = \frac{2}{w+1}$$

Note: $\mathbb{E}(X)$ for 0 mod p method can be derived as $\frac{1 + ln(w))}{w}$

F. Locality
1. Locality:
   a) Let S be a selection function taking a w-tuple of hashes and returning an integer $i \in [1..w]$. A fingerprinting algorithm is $local$, if for every window $h_i \ldots h_{i+w-1}$ the hash at position $i + S(h_i \ldots h_{i+w-1})$ is selected as fingerprint.
2. $\Rightarrow$ winnowing is a local fingerprinting method
3. Question: What would be an example for a non-local fingerprinting method?
   a) Take every 50th hash as fingerprint.

G. Applicagtions
   1. The fingerprinting method of winnowing has a wide range of practical applications:
      a) Sequencing binning as a preprocessing for the assembly of metagenomes
      b) suffix array sampling
      c) annotation of metagenmics read
   2. Note 1: Winnowing was independently developed by another group under the name “minimisers”

V. Minimal hashing

A. The concept of minimal hashing can be seen as a generalisation of the winnowing principle.

B. Idea: Similar to fingerprinting, the idea is to compute a sketch of a sequence but also use it for distance estimation.

C. Approach: Given a text T, compute a hash value for every consecutive k-mer in T and retain a set S of the smallest hashes. This set is called the **bottom sketch** of size s, where $s = |S|$

D. Probability of a random k-mer K mapping to a text $T = t_1 \dots t_n, t_i \in [1..\sigma]$ is

$$P(K \in T) = 1 - \left(1 - \frac{1}{\sigma^k}\right)^n$$

$\Rightarrow$ we need to choose k, s.t. $P(K \in T)$ is small.

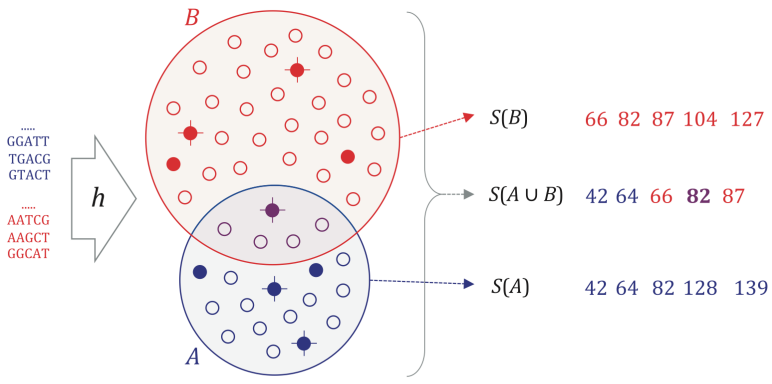
E. Distance
   1. We can use the sketches as an unbiased estimate of the k-mer population of two strings (genomes).
   2. Jaccard Index: Given k-mer sets A and B, the **Jaccard index (Jaccard similarity)** J is defined as

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

   3. We can estimate this using the sketches $S(A)$ and $S(B)$:

$$J(A,B) \approx \frac{|S(A) \cap S(B)|}{|S(A) \cup S(B)|}$$

   4. with an error bound of $\epsilon = O\left(\frac{1}{\sqrt{s}}\right)$, where s is the sketch size.



adapted from [Ondc

AlgoPop – Dr. Andre Kahles

F. MASH Distance
1. in their original paper Ondov et al. estimate a distance that correlates well to the often used measure of *average nucleotide identity (ANI).*
2. Under a Poisson model and a given mutation probability d, the probability of no mutation in a k-mer is e^{-kd}
3. On a genome perspective this should correspond to the expected fraction of conserved k-mers w over all k-mers t and hence:
$$e^{-kd} = \frac{w}{t} \Rightarrow d = -\frac{1}{k} In \frac{w}{t}$$
4. If we use the average genome size n instead of t and use the expression of the Jaccard value j as $\frac{w}{2n - w}$, we can derive the MASH distance
$$D = -\frac{1}{k} \ln \frac{2j}{1 + j}$$

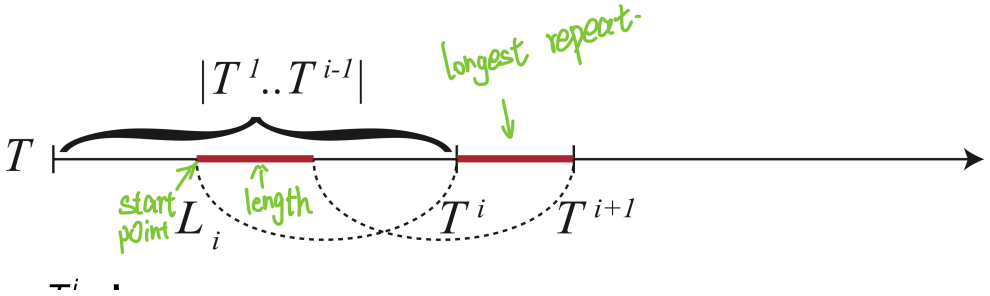
VI. Example Exam Questions
A. Which method compute a sketch of a given (long) string and what is it used for?
1. min-hashing;
2. It uses bottom sketch to calculate the fingerprint of one string and is used to calculate for example the distance between two strings.

Lecture 8: Genome compression

I. Motivation
   A. Given a large number of sequenced genomes, the task to store them efficiently becomes more relevant
   B. For human, the amount of variation observed between any two individuals is less than 0.1% of all positions
   C. This leaves room for space gains through compression strategies, some of which we have already discussed.

II. Compression Strategies
   A. Some possible strategies have already been discussed or are easy to derive from data structures presented previously
   B. Reference-based compression: picks one of the genomes as refrence and expresses all others as lists of edit operations (alignment based)
   C. String graph compression: picks one genome as reference path and expresses all others as alternate path. Haplotype encoding with pBWT
   D. Storage in a succinct k-mer graph: stores the set of substrings of length k in a de-bruijn graph. Needs additional information to unambiguously reconstruct genomes.
   E. Plain text compression is still one of the most frequently used techniques to represent genomics data in small space

III. Lempel-Ziv parsing
   A. Concept
      1. Given a text $T = t_1 t_2, \ldots t_n \in \Sigma^n$, then we seek to obtain a partitioning $T = T^1 T^2 \ldots T^p$ such that $\forall i \in [1..p]$ T^i will be the longest prefix of $t_{|T^1..T^{i-1}|+1} \ldots t_n$ which has an occurrence starting anywhere in $T^1 .. T^{i-1}$



      2. We call the strings T^i **phrases**
      3. using only $2p \log n$ bits of space, string T can now be encoded as a list of phrases $(|T^1|, L_1), (|T^2|, L_2), \ldots, (|T^p|, L^p)$. The list can be decoded into T in $O(n)$ time.
   B. Longest previous factor
      1. At the heart of Lempel-Ziv parsing lies the problem to find the longest previous factor
      2. **Longest previous factor**: Given a string T, the longest previous factor at position a in T is the longest prefix of $t_a .. t_n$ that has an occurrence in T starting at a position in $[1..a-1]$

3. We will now show that a naive encoding of a Lempel-Ziv parsing takes up $O(nlog\sigma)$ space.

C. Space of Lempel-Ziv parsing
1. Lemma 6.1 :
   a) Number of phrases I
      (1) The maximum number of phrases in a text T of length n over alphabet $[1..\sigma]$ is $\frac{2n}{\log_\sigma n} + \sigma\sqrt{n}$
   b) Proof:
      (1) The lemma is trivially true for $\sigma > \sqrt{n}$
         (a) worst case: every letter of text T is a phrase and since $\sigma > \sqrt{n}$, we have $\sigma\sqrt{n} > (\sqrt{n})^2 \Rightarrow \sigma\sqrt{n} > n$
      (2) For $\sigma < \sqrt{n}$ fix $b = \lceil \log n / 2log\sigma \rceil = \lceil \log_\sigma^n / 2 \rceil$, then
         (a) The upper bound for distinct substrings of length b is $\sigma^b = \sigma^{\frac{\log_\sigma n}{2}} = \sqrt{n} < \sigma\sqrt{n}$
         (b) The number of phrases shorter than b is upper bounded by $\sigma\sqrt{n}$
         (c) The number of phrases at least as long as b is at most $n/b = n/(\frac{\log_\sigma n}{2}) = 2n/\log_\sigma n$
      (3) Hence, the total space for all phrases is at most $2n/\log_\sigma n + \sigma\sqrt{n}$
2. Lemma 6. 2 : Number of phrases II
   a) The maximum number of phrases in a text T of length n over alphabet $[1..\sigma]$ is upper bounded by $3n/\log_\sigma n$.
   b) Proof: Given alphabet size σ, we can distinguish two cases
      (1) $\sigma < \sqrt[3]{n}$: by Lemma 6.1, we substitute for σ and get
      $$\frac{2n}{\log_\sigma n} + \sigma\sqrt{n} < \frac{2n}{\log_\sigma n} + n^{\frac{1}{3}+\frac{1}{2}} = \frac{2n}{\log_\sigma n} + n^{\frac{5}{6}} < 3n/\log_\sigma n\,.$$
      (2) $\sigma \geq \sqrt[3]{n} \Rightarrow 3 \geq \log_\sigma n \Rightarrow n = \frac{3n}{3} \leq \frac{3n}{\log_\sigma n}$
   c) For p phrases, we need $2p \log n$ bits. Hence, we need at most $\frac{6n}{\log_\sigma n} \log n = \frac{6n}{\frac{\log n}{\log \sigma}} \log n = 6n \log \sigma \in O(n \log \sigma)$ bits.

D. Basic algorithm
1. Lemma 6.3 - Basic Lempel-Ziv
   a) The Lempel-Ziv parse of $T = t_1 t_2 \cdots t_n \in [1..\sigma]$ can be computed in $O(n \log \sigma)$ time and $O(n \log n)$ space. This includes generating all pairs $(|T^1|, L_1), (|T^2|, L_2), \ldots, (|T^p|, L^p)$
2. Algorithm (sketch):
   a) Assume $T^1 T^2 .. T^{i-1}$ has been built and we want to find the next phrase T^i starting at $a = |T^1 .. T^{i-1}| + 1$

b) Assume we have built an augmented suffix tree ST_T on T that stores at each internal node v a value N_v describing the minimum starting position of a suffix under v. (Built in linear time with depth-first traversal)

c) Find the deepest interval node v s.t. its path is a prefix of $t_a \,..\, t_n$ and $N_v < a$. Then set L_i to N_v and assign the length of the path to a to $|T^i|$. Update a as $a = a + |T^i|$

3. Complexity

a) each step in the tree takes $O(\log \sigma)$ and hence each phrase takes $O(|T^i| \log \sigma)$ time

(1) if the suffix tree is represented using Bi-directional BWT and wavelet tree

b) $\sum_i |T^i| = n \Rightarrow O(n \log \sigma)$ time

c) suffix tree needs $O(n \log n)$ bits to store $O(n)$ values N_v

d) $\Rightarrow O(n \log \sigma)$ time and $O(n \log n)$ space

E. Space efficient algorithm

1. lemma 6.4 - space -efficient Lempel-Ziv

a) We can compute the Lempel-Ziv parsing $T^1 T^2 \,..\, T^p$ of a given text $T = t_1 t_2 \,..\, t_n$, $t_i \in [1..\sigma]$ in $O(n \log n \log^2 \sigma)$ time and $O(n \log \sigma)$ space, including generating all pairs $(|T^1|, L_1), (|T^2|, L_2), \ldots, (|T^p|, L^p)$

2. Before presenting the algorithm, we introduce auxiliary data structures

a) a succinct suffix array $BWT_{T\#}$ if T with a sampling rate of $r = \log n$

(1) $n \log \sigma(1 + o(1))$ space for BWT in wavelet tree

(2) $(n/\log n) \times \log n = O(n)$ bits for storing r SA values

b) an array $R[1..n/\log n]$ to store at position i the minimum of the i-th SA range. This costs $O(n)$ space.

$$R[i] = \min(SA_{T\#}[(i-1)\log n + 1..i \log n])$$

c) an Range Minimum Query data structure on R returning $\min_{i \in [l..r]} R[i]$ for range $[l \,..\, r]$

3. Algorithm: Assume we already generated $T^1 T^2 \,..\, T^{i-1}$ and now look at position $a = |T^1 T^2 \,..\, T^{i-1}| + 1$ in T. Using $BWT_{T\#}$, we can compute position $j = SA^{-1}_{T\#}[a]$ (s.t. $SA_{T\#}[j] = a$) in $O(\log \sigma)$ per step.

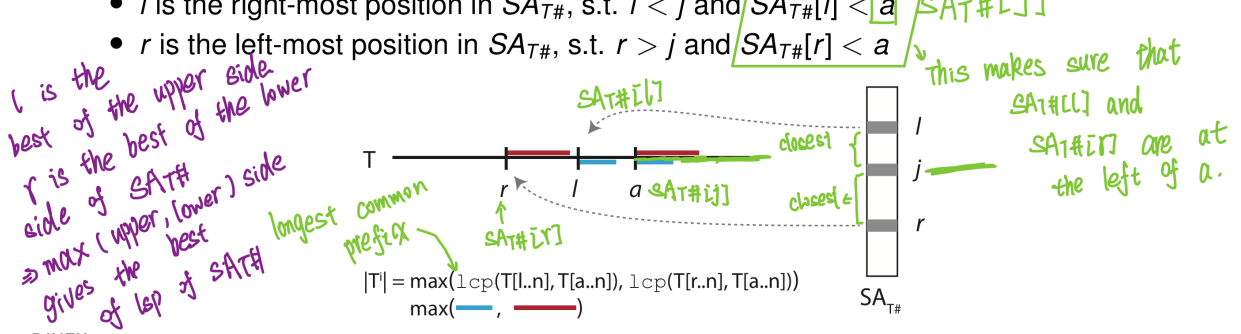
4. We now need to search for two positions l and r such that:

a) l is the right-most position in $SA_{T\#}$, s.t. $l < j$ and $SA_{T\#}[l] < a$

b) r is the left-most position in $SA_{T\#}$, s.t. $r > j$ and $SA_{T\#}[r] < a$

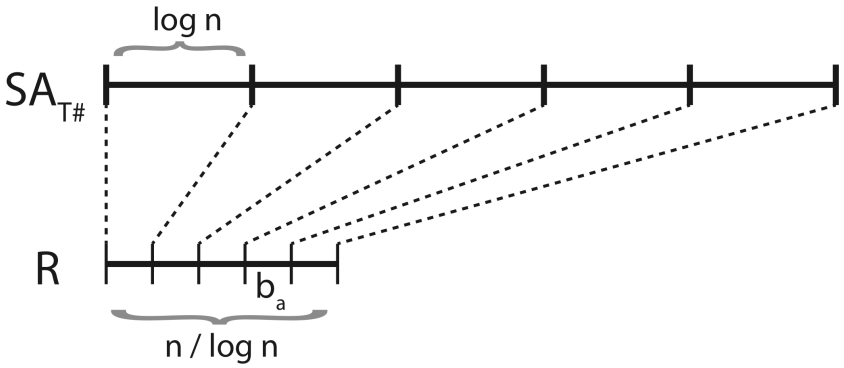
We now need to search for two positions *l* and *r*, such that:

- *l* is the right-most position in $SA_{T\#}$, s.t. $l < j$ and $SA_{T\#}[l] < a$
- *r* is the left-most position in $SA_{T\#}$, s.t. $r > j$ and $SA_{T\#}[r] < a$



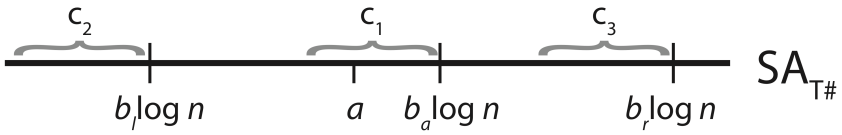
F. Finding l and r efficiently

1. We will utilise auxiliary array $R[1..\frac{n}{\log n}]$ containing the min of the SA blocks
2. Let $b_a = \lceil a/\log n \rceil$. Then use binary search on R to find b_l and b_r such that
   a) b_l is the right-most position in $R[1..b_a - 1]$ s.t. $b_l < b_a$ and $R[b_l] < a$
   b) b_r is the left-most position in $R[b_a + 1..\frac{n}{\log n}]$ s.t. $b_r > b_a$ and $R[b_r] < a$



3. Using the RMQ built on R, this takes $O(\log n)$ time for each step of binary search, thus $O(\log^2 n)$ for the whole search
4. From b_r and b_l, we can derive candidate regions for finding the next phrase:

$$c_1 = SA_{T\#}[(b_a - 1)\log n + 1..b_a \log n]$$
$$c_2 = SA_{T\#}[(b_l - 1)\log n + 1..b_l \log n]$$
$$c_2 = SA_{T\#}[(b_r - 1)\log n + 1..b_r \log n]$$



   a) Search right half of c_1 (scan $SA_{T\#}[a + 1..b_a \log n]$) for $a < r \leq b_a \log n$ and $SA_{T\#}[r] < a$; If not found scan for the left-most r in c_3
   b) Search left half of c_1 (scan $SA_{T\#}[(b_a - 1)\log n + 1..a - 1]$) for $(b_a - 1)\log n + 1 \leq r < a$ and $SA_{T\#}[l] < a$; If not found scan for the right-most l in c_2

G. Completing the search

1. Extracting the $SA_{T\#}$ positions takes $O(\log n \log \sigma)$ per position. For 3logn positions total time is $O(\log^2 n \log \sigma)$
2. Finally, we can use l and r to complete the search and compute
   $I_1 = \text{lcp}(T[SA_{T\#[l]}..n], T[a..n])$
   $I_2 = \text{lcp}(T[SA_{T\#[r]}..n], T[a..n])$
3. We can compute this in $O((I_1 + I_2)\log \sigma)$

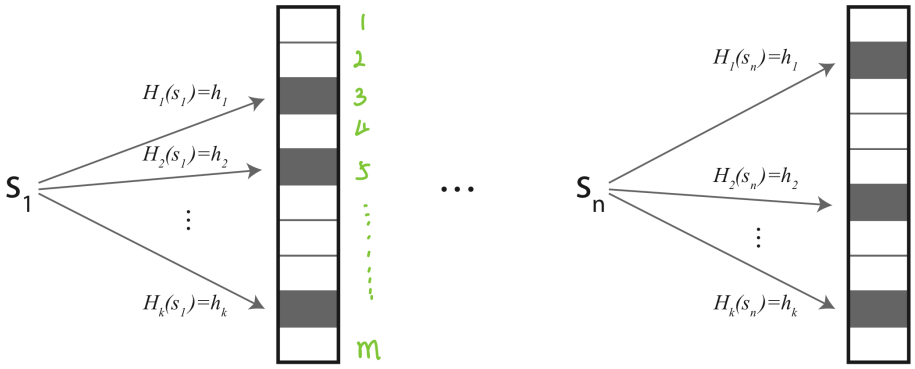
4. Set $|T^i| = \max(|I_1|, |I_2|)$ and let $L_i = \begin{cases} l, & \text{if } |I_1| > |I_2| \\ r, & \text{else} \end{cases}$

IV. Example Question
   A. Describe what the term *phrase* means in Lempel-Ziv parsing and what such a phrase would be used for
      1. In Lempel-Ziv parsing we partition the string into a sequence of substrings $T = T_1 T_2 \dots T_n$ where for each substring T_i is the longest prefix of string $T_{i-1} + 1..T_n$ which is a substring in string $T_1 \dots T_{i-1}$
      2. this phrase can then be used for compressing the text, since we just need to store where it starts in the previous string and how long the phrase is.

Lecture 9 - Approximate genome representation

I. Approximate membership data structures
   A. We have looked into succinct but exact representation of strings and graphs.
   B. For many applications an inexact representation with a small error probability is acceptable
   C. Idea: Further reduction of space through lossy compression employing probabilistic data structures
   D. Approximate Membership Query (AMQ) data structure
      1. An **approximate membership query** (AMQ) data structure is a data structure that supports the dictionary operations *insert, lookup* and (optimally) *delete* efficiently, providing a correct answer to queries with high probability.

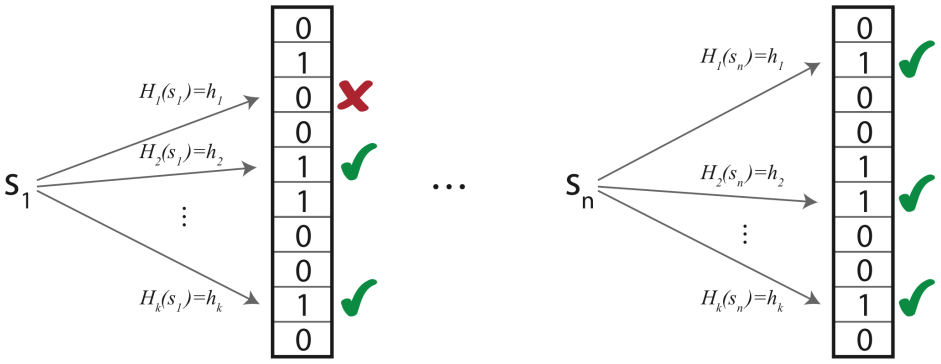
II. Bloom Filters
   A. A Bloom filter (BF) is an early example of Approximate Membership Query(AMQ). It was already suggested in the 1970s by Burton Bloom.
   B. A Bloom Filter(BF) consists of the following elements:
      1. a bit-array $\mathcal{F} = \{0,1\}^m$ initialised with all 0
      2. an associated set of hash functions $\{H_1, \dots, H_k\}$ that maps elements from a given set to the range $[1..m]$
      3. for the context of this lecture, we consider $H_i : \Sigma^n \mapsto [1..m]$, where Σ is a given input alphabet
   C. A classical Bloom Filter, supports the operations *insert* and *query*
   D. Insert
      1. Given an element $s \in \Sigma^n$, we compute all associated hash values for s as $h_1, \dots, h_k$, with $h_i = H_i(s)$. Then we set $\mathcal{F}[h_i] = 1 \quad \forall i \in [1,..,k]$.



E. query

1. The *query* operation is the trivial inversion of *insert.* For a given query string $s \in \Sigma^n$, all hash values $h_1, \ldots, h_k$ with $h_i = H_i(s)$ are generated.
2. $membership(s) = \begin{cases} 1, & \text{if } \mathcal{F}[h_i] = 1 \ \forall i \in [1..k] \\ 0, & \text{else.} \end{cases}$
3. Note: this leads to a one-sided error
   a) since we can only generate false positives because when the membership(s) returns 1, it can be the result of the combination of several different s_is
   b) But if the membership(s) returns 0, it is always true

with $h_i = H_i(s)$ are generated.



F. False positive rate

1. For practical applications, we are interested to bound the one-sided error.
2. Given k (perfectly random) hash functions and a total of n elements to be inserted into a filter of size m, the probability for a bit being still 0 is:

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

3. Let ρ be the proportion of 0 bits after all n elements are inserted. Then $\mathbb{E}(\rho) = p$. The probability f for a false positive is then

$$f = (1-\rho)^k \simeq (1-p)^k$$

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

$$\simeq (1 - e^{-\frac{kn}{m}})^k = (1 - e^{-\frac{k}{r}})^k$$

where $1/r = n/m$ is the occupancy of the filter.

4. Using

$$e^{\ln\left(\left(1-e^{-\frac{k}{r}}\right)^k\right)} = e^{k\ln\left(1-e^{-\frac{k}{r}}\right)}$$

5. We can directly minimise

$$g = k\ln(1 - e^{-\frac{k}{r}})$$

6. Computing the first derivative

$$\frac{\partial g}{\partial k} = \ln\left(1 - e^{-\frac{k}{r}}\right) + \frac{k}{r}\frac{e^{-\frac{k}{r}}}{1 - e^{-\frac{k}{r}}}$$

G. Optimal Size
   1. The derivative of g with regard to k is 0 when $l = \ln 2 \frac{m}{n} \simeq 0.71r$
   2. If we choose the optimal number of k hash functions, then the minimal false positive rate evaluates to
$$f = \left(1 - e^{-\ln 2 r/r}\right)^{\ln 2 r} = \frac{1}{2}^{\ln 2 r} \simeq 0.6185^r$$
   3. Using the optimal k, we can also solve for m and get the expected size of the Bloom filter, given a desired false positive rate f
$$m \simeq 1.44 \log_2 \left(\frac{1}{f}\right) n$$

H. Use cases
   1. There are multiple practical use cases for bloom filters in bioinformatics
   2. For streaming applications, e.g. k-mer counting, to identify non-noise (solid) k-mers (only retain k-mers that occur at least twice)
   3. sequence classification, where each class represents all its members through a Bloom filter
   4. Representation of probabilistic de Bruijn Graphs
      a) hash all node labels (k-mers) of the graph into a Bloom Filter
      b) traversal through querying the desired / all possible neighbours (might lead to false positive edges)
   5. We will further discuss the approximate de Bruijn graph idea

III. Exact de Bruijn graphs using Bloom filters

A. Idea: Use an additional data structure to store a set C of critical false positive values that lead to false positive branching in the graph.

B. Given the set S of all true positives, then let $\mathscr{E}$ be the set of all valid extensions in the graph from values in S
   1. $\mathscr{E} \cap \{\text{all false positives}\} = \{\text{critical false positives}\}$

C. If P is the subset of elements in ε for which the Bloom filter answers “Yes”, then we can properly define the set of critical false positives as $C = P \backslash S$

D. Observation: There is a trade-off between the size of the Bloom filter (and thus the false positive rate) and the size of C
   1. if the bloom filter is large, the false positive rate is small, and thus the size of C is small
   2. if the bloom filter is small, the false positive rate is large, and thus the size of C is large

E. Goal: Minimise the total storage size
   1. total storage size takes the size of the bloom filter and the size of set C into account

F. Optimising the set of critical false positives
   1. Observation: As we are dealing with de Bruijn graphs, each node in S can have at most 2σ many neighbours
      a) for genome de Bruijn graphs, it would be 4 incoming node and 4 outgoing node.
   2. This leads to the following insights:
      a) the expected size of C is $2\sigma n f$
         (1) n is the number of nodes, f is the false positive rate

b) if we store each k-mer in C explicitly, we need in total $\log(\sigma)k \cdot 2\sigma$ bits
c) together with the bloom filter, this results in a storage cost of

$$1.44 \log_2 \left(\frac{1}{f}\right) n + 2 \log_2(\sigma) k \sigma n f \text{ bits}$$

d) This becomes minimal for $f = \frac{1.04}{\log_2(\sigma)k\sigma}$

IV. Cascading Bloom Filters
A. Question: Can we further reduce the storage requirement for critical false positives?
B. Idea: Instead of storing critical false positives explicitly, use another Bloom filter to store them approximately. Deal with new false positives the same way, recursively.
C. Approach:
1. Let T_0 be the set of all k-mers occurring in a text T. We store all elements of T_0 in a Bloom filter B_1 . Let T_1 be the set of critical false positives in B_1
2. For all further steps, represent T_i with a new Bloom filter B_{i+1} and a set of critical false positives T_{i+1}
D. Correctness
1. Note: We assume that only k-mers $u \in T_0$ and their direct neighbours in the graph will ever be queried.
2. As T_{i+2} will store the false false positives of T_i, it holds that:

$$T_0 \supseteq T_2 \supseteq \ldots \supseteq T_{2i} \text{ and } T_1 \supseteq T_3 \supseteq \ldots \supseteq T_{2i+1}$$

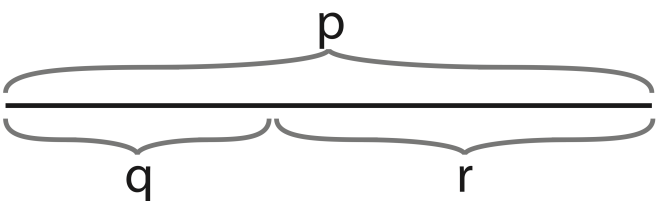
3. Observation: $u \notin B_{i+1} \rightarrow u \notin T_i$ (due to one-sided error of BF)
4. Lemma : Consider the smallest i such that $u \notin B_{i+1}$ then it holds that
a) if i is odd $\Rightarrow u \in T_0$
b) if i is even $\Rightarrow u \notin T_0$
5. Proof: We will conduct the proof via induction on i.
a) Base cases:
(1) $i = 0 : u \notin B_1 \Rightarrow u \notin T_0$
(2) $i = 1 : u \in B_1 \wedge u \notin B_2 \Rightarrow u \notin T_1$(false false positive) $\Rightarrow u \in T_0$
b) Induction step $(i \geq 2)$:
(1) $u \in B_1 \cap B_2 \cap \ldots \cap B_i \Rightarrow u \in T_{i-1} \cup T_i$
(2) $u \notin B_{i+1} \Rightarrow u \notin T_i$
c) From 1) and 2) it follows that $u \in T_{i-1}$
d) Since $T_{i-1} \subseteq T_0$ for odd i and $T_{i-1} \subseteq T_1$ for even i, the Lemma follows.

V. Bloom Filter Alternatives
A. Despite their useful features, Bloom filters have a row of disadvantages
1. The operation *delete* is not supported, hence once added, elements can not be removed
2. can be held in RAM but are not very cache friendly due to random access patterns; scale poorly out of RAM
3. cannot be dynamically resized
4. query time depends on the number of hash functions being used
B. We will further explore alternatives, that address these shortcomings

VI. Quotient Filters

A. Quotient filters rely on the idea of *quotienting*:

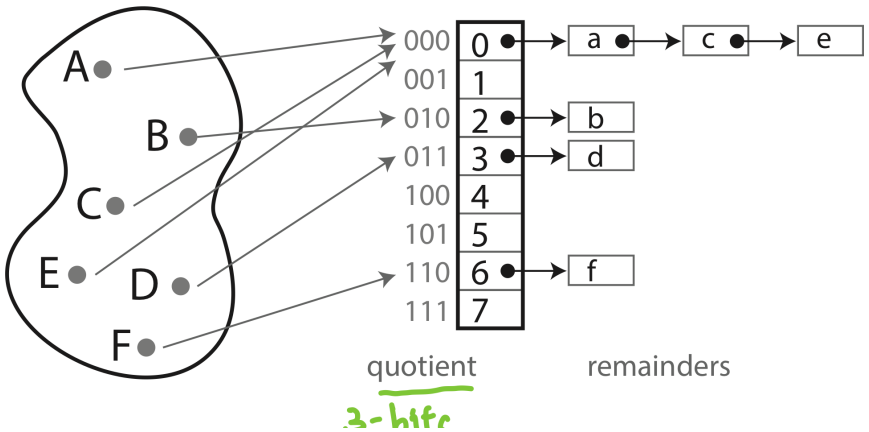
1. For a given element, compute a p-bit hash/fingerprint
2. split the fingerprint into the q most significant bits (the **quotient**) and the r least significant bits (the **remainder**), where $p = q + r$



B. Approach: We will use the quotient to guide storage of the remainder, only implicitly representing the quotient

C. Schema

1. Idea: Store the remainder of a hash in a bucket indexed by the quotient



f	f_q	f_r
A	0	a
B	2	b
C	0	c
D	3	d
E	0	e
F	6	f

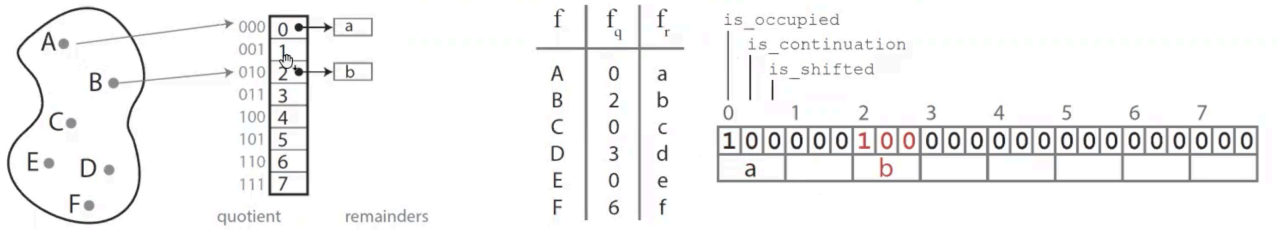
2. If two elements have the same quotient this leads to a *soft collision:*
   a) in case of collision, remainders are assigned to bucket in sorted order (leading to occasional shifts of elements)
   b) the original slot assigned to each remainder is called **canonical slot**
   c) all remainders with the same quotient are stored consecutively (forming a **run**)
   d) a **cluster** is a maximal sequence of occupied slots whose first element is the only element in the cluster residing in its canonical slot.

D. Storage organisation

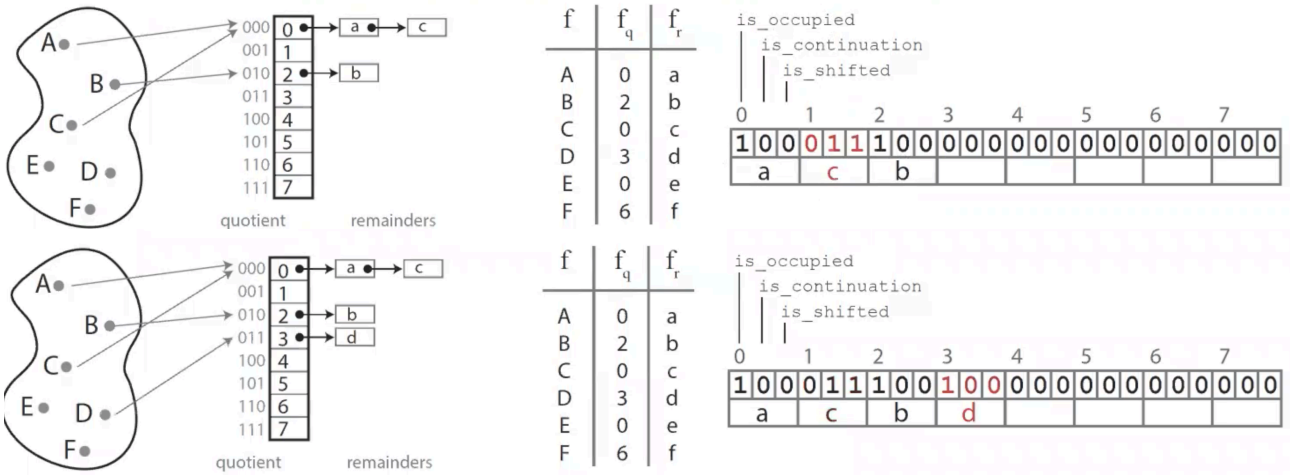
1. The fingerprint lists of each bucket are stored continuously in memory. We use three additional bits per slot to recover the full fingerprint.
   a) $is\_occupied$: set to 1, if some stored element has this slot as its canonical slot; else 0
   b) $is\_continuation$: set 1 if the stored element is part of a run but not the leading element; else 0;
   c) $is\_shifted$: set to 1 if the stored element is not in its canonical slot; else 0
2. insert operation:
   a) mark the slot assigned to the quotient as occupied
   b) shift any colliding remainders forward (if necessary) and while doing so update the bits for $is\_continuantion$ and $is\_shifted$

c) Example

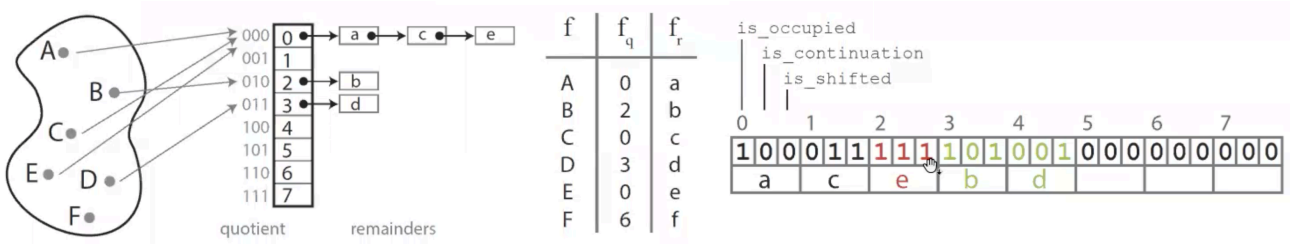
Based on the previous example, we show the storage organisation of the elements



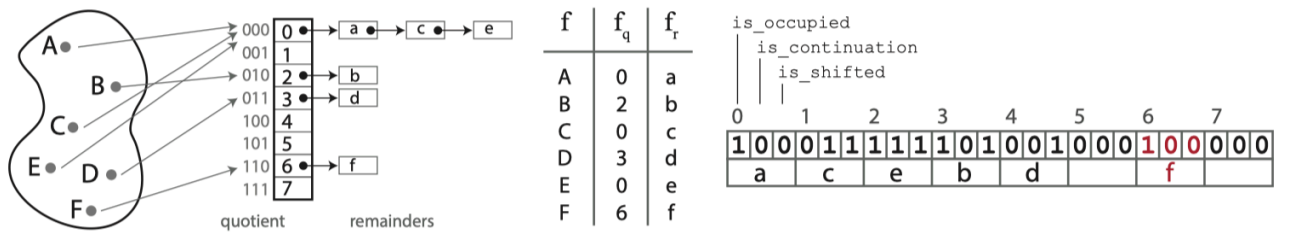
Single element insertion just marks the $is\_occupied$ bit



in case of soft collisions, start a run and shift the element to the neighbouring slot



if a slot is already occupied, subsequently shift the neighbouring elements further



E. False positives
   1. Note: False positives can only occur, for identical fingerprints (shared quotient and remainder!)
   2. Let the load factor of the hash-table be $\alpha = \frac{n}{m}$, with ($m = 2^q$) where n elements are hashed into 2^q slots.
   3. This results in a false positive rate f of:
$$f = 1 - e^{-\frac{\alpha}{2^r}} \leq \frac{1}{2^r}$$
   4. Note: For a fixed number of hashed elements, and a fixed false positive rate, a QF with $\alpha = \frac{3}{4}$ requires 20% more space than the equivalent Bloom Filter with 10 hash functions

VII. QF vs BF
A. Despite their slightly increased memory footprint, quotient filters have some advantages over Bloom filters
   1. elements can be deleted from the filter
   2. merging of multiple filters without re-hashing all the elements
   3. better locality than Bloom filters and thus higher cache efficiency
B. As *insertion, deletion* and *lookup* require traversal of runs, one needs to upper-bound their length distribution
   1. with high probability in $O(\log m) \sim O(q)$

VIII. Rank-Select Quotient Filters (RSQF)
A. The metadata overhead of QFs is 3 bits per storage slot. The encoded information essentially summarises two properties of a given slot i:
   1. $in\_use(i)$: there is a remainder stored at slot i
   2. $occupied(i) : \exists x$ with $h(x) = i$
B. Observations: Remainders with the same quotient are stored in consecutive slots (in sorted order), forming a run
C. Idea: Use only 2 bit-vectors to encode metadata:
   1. $occupieds$: marks all occupied slots
   2. $runends$: marks the last element of each run
D. Note:
   1. there is a 1:1 match between the set bits in $occupieds$ and $runends$
   2. we can use $rank()$ and $select()$ on the bit vectors for navigation
E. Operations
   1. Let q and r be quotient and remainder of a fingerprint x, respectively
   2. Lookup:
      a) $occupieds[q] = 0 \Rightarrow x \notin QF$
      b) $occupieds[q] = 1$
         (1) get number of occupied slots up to q using $t = rank_1(q, occupieds)$
         (2) get corresponding runners bit with $select_1(t, runends)$
         (3) walk back until x is found, next $runends$ bit is set, or slot q is reached
   3. Insert:
      a) if $rank_1(q, occupieds) < q \Rightarrow$ slot q is not in use $\Rightarrow$ insert x in slot q
      b) else, find the next free slot and shift occupying elements as necessary

F. Efficiency
   1. The RSQF only needs 2 bits metadata per slot, but in the worst case $O(n)$ time for $rank()$ and $select()$
   2. Question: Why can't we use our classical constant-time rank and select implementation?
      a) Because we need to maintain auxiliary data structure
   3. Ideas to increase efficiency:
      a) Store additional offset value for $runends$. When using an 8 bit counter to summarise groups of 64 slots, one can decrease memory overhead to 0.125 bits per slot while increasing efficiency
      b) storing $occupieds, runends,$ and $offset$ in blocks, increases cache efficiency

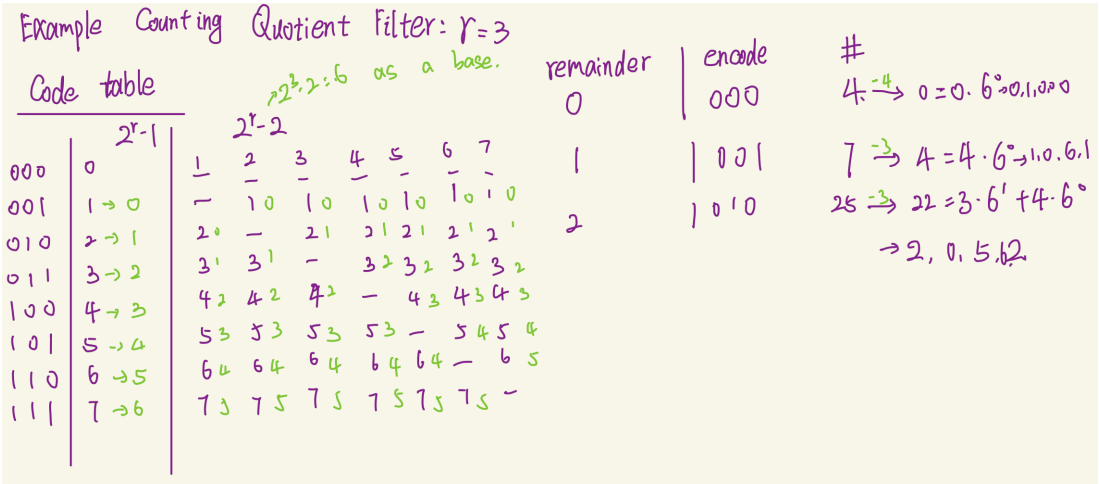
IX. Cuckoo Hashing
   A. As a prelude for the next filter, we need to look into cuckoo hashing
   B. Idea: A cuckoo hash table consists of an array of buckets and elements are inserted into said buckets.
   C. Approach: (to insert element x)
      1. assign x to candidate buckets using hash functions $h_1(x)$ and $h_2(x)$
      2. this leads to two possible cases
         a) If either bucket is empty $\Rightarrow$ Insert x into the empty one
         b) If no bucket is free, choose one of the two, insert x and take another item from that bucket to move it ti its alternate location (and keep moving until a free bucket is found)
   D. Problem: For this to work, one needs to store the original values to find the relocation bucket through re-hashing.
   E. Partial-key Cuckoo Hashing
      1. Idea: Instead of using the full elements, we can use part of the fingerprint to determine the alternate location.
      2. Modify the hash functions as follows:
$$h_1(x) = hash(x)$$
$$h_2(x) = h_1(x) \oplus hash(\mathcal{F}(x))$$
      where $\mathcal{F}(x)$ is the fingerprint computed from x. (for a uniform distribution)
      3. Observation: if $\oplus$ corresponds to XOR, it follows that
$$h_2(x) \oplus hash(\mathcal{F}(x)) = h_1(x)$$
      4. This has a very convenient consequence: Given bucket i it is straightforward to compute the alternative bucket j as:
$$j = i \oplus hash(\mathcal{F}(x))$$

X. Counting Quotient Filters (CQF)
   A. Counting Quotient Filters are an extension of RSQFs discussed earlier
   B. Observation: Quotient Filters store all remainders in unary encoding (storing one copy of the remainder for each occurrence).
   C. idea: Re-purpose some slots in the filter to store counters instead of remainders.
   D. Approach:
      1. Use escape sequences to discern between counters and remainders
      2. do not use counters for remainders occurring less than 3 times
      3. use the break of the strictly non-decreasing pattern of remainders inside a run to make counters
      4. counter slots are delimited by the same remainder slot on both ends

E. Coding:

Count	Encoding	Rules
$C = 1$	x	none
$C = 2$	x, x	none
$C > 2$	$x, c_{\ell-1}, \ldots, c_0, x$	$x > 0$ $c_{\ell-1} < x$ $\forall i\ c_i \neq x$ $\forall i < \ell - 1\ \ c_i \neq 0$
$C = 3$	$0, 0, 0$	$x = 0$
$C > 3$	$0, c_{\ell-1}, \ldots, c_0, 0, 0$	$x = 0$ $\forall i\ c_i \neq 0$

1. We cannot allow 0 or x in the encoding of the count, as they are used as delimiters.
2. $\Rightarrow$ As we know that count ≥ 3, we encode it as $c_{\ell-1}, \ldots, c_0$ in base $2^r - 2$, using symbols $1,2,...,x-1,x+1,...,2^r-1$. We prepend a 0 if $c_{\ell-1} > x$



XI. Sample Questions

A. Why do Bloom Filters have a “one-sided error”?

1. Because bloom filters would think an element is in the bloom filter if and only if all k hashed bit is set to 1. So, if one element is not in the bloom filter, it would have to be that at least 1 of the k hashed bit would be 0, thus bloom filter can only have true negatives.
2. It can have false positives because the k hashed bits of 1 may come as a combination of some existing elements

B. What consequences would it have for a de Bruijn graph that is stored in a Bloom filter, if an answer to a query is a false positive?

1. This would lead to a false positive edge in the de Bruijn graph, meaning we would find an edge that does not exist in the de Bruijn graph

Lecture 10: Cardinality estimation

I. Cardinality estimation-Intro
  A. Cardinality:
    1. Definition: Given a multiset M of elements drawn from a joint universe $\mathcal{U}$, the cardinality $C(M)$ describes the number of distinct elements in M
    2. Goal: Given a multiset M estimate its cardinality $C(M)$ using minimal space. (That is, $\ll$ than $|M|$ or even $C(M)$
    3. Trivial solution: Stream through M and keep a sorted list of all distinct elements $\Rightarrow$needs $O(C(M)\log C(M))$ space
    4. What are the alternatives?
II. Sampling-based approaches
  A. Subsampling
    1. Idea: From the given multiset M, randomly select a subset $\bar{M}$ and then proceed as follows:
      a) compute $C(\bar{M})$
      b) estimate $C(M)$ as $C(\bar{M}) \cdot |M|/|\bar{M}|$
    2. Problem: Does not work well for very skewed distributions of element-counts in M and is not stable
      a) Can lead to drastic mis-estimation of cardinality

Example (cardinality 10): (Example of skewed distribution)

```
M:   0,0,0,0,0,0,0,0,0,0,0,1,2,3,4,5,6,7,8,9
M*:      x   x x x x
M**:     x                          x x x x
```

$M* \Rightarrow C(M) = 1 \cdot 20/5 = 4$ $\quad M** \Rightarrow C(M) = 5 \cdot 20/5 = 20$

  B. Linear Counting
    1. Idea: Use a hashing function to estimate cardinality
    2. Approach: Given a multiset M, for each element $x \in M$, compute a hash value $h(x)$ using a random hashing function $h : D \mapsto [1...m]$. Initialise an array $BITMAP[1...m]$ to 0.
      a) For each $x \in M$
        (1) compute $i = h(x)$
        (2) set BITMAP[i] = 1
    3. Math
      a) Probability that a bit in BITMAP is unset after inserting n items:

$$P(BITMAP_j = 0) = (1 - \frac{1}{m})^n$$

      b) Expectation for the number of unset bits m_0 can be estimated as:

$$E(m_0) = m(1 - \frac{1}{m})^n \simeq me^{-\frac{n}{m}}$$

c) estimate cordiality as $\hat{n} = -m \ln \frac{m_0}{m}$, where $\frac{m_0}{m}$ is the fraction of 0 bits in BITMAP

4. Problem:
   a) To increase accuracy for large datasets, the hash table needs to grow linearly in the order of the cardinality of the set to be estimated

III. Flajolet-Martin Counting

A. Cardinality observables

1. An alternative set of strategies are based on **observables** of cardinality
2. Cardinality observable: Given a multiset M over a set of elements S, an **observable** of M is a function that only depends on the underlying set S, independent of any replications in M.
3. We distinguish two main classes of observables:
   a) Order statistics observables: based on order statistics, e.g. $X = \min(S)$, where the assumption is that $|S|$ is of the order of $1/X$
   b) Bit-pattern observables: based on the probabilities of certain patterns in the binary representation of values of S

B. Flajolet-Martin Counting

1. Let $h(x)$ be a (sufficiently uniform hash function h that maps into the range of numbers that can be represented with L bits

$$h : \mathscr{D} \mapsto [0...2^L - 1]$$

2. We define a function $bit(y, k)$ that corresponds to the k-th most significant bit in the binary representation of y, such that

$$y = \sum_{k \geq 0} bit(y, k)2^k$$

3. We further define $\rho(y)$ as a function returning the position of the most-significant 1-bit in the binary representation of y:

$$\rho(y) = \begin{cases} \min_{k \geq 0} bit(y, k) \neq 0 & \text{if y > 0} \\ L & \text{else} \end{cases}$$

4. Approach:
   a) Observation: Under the assumption of uniform distribution of values of $h(x)$, a substring 0^k1 appears with probability $2^{-(k+1)} = 2^{-k-1}$
   b) Idea: Record observations of such a pattern in an array $BITMAP[0..L-1]$
   c) Approach:
      (1) initialise $BITMAP$ as all 0
      (2) For all $x \in M$ do:
         (a) $index = \rho(h(x))$
         (b) $BITMAP[index] = 1$

   $\Rightarrow BITMAP[i]$ will only contain 1, iff a pattern 0^i1 was observed among the hashed values of members of M

d) Observation: if n is the number of distinct elements in M, then
   (1) $BITMAP[0]$ is accessed $\simeq n/2$ times $\Rightarrow$ half of the elements would have 0 leading 0
   (2) $BITMAP[1]$ is accessed $\simeq n/4$ times $\Rightarrow \frac{1}{4}$ of the elements would have 1 leading 0
   (3) …
   (4) $BITMAP[i]$ is accessed $\simeq n/2^{i+1}$ times

e) This leads to the following **expectation**:
   (1) if $i \gg \log_2(n)$, $BITMAP[i]$ is almost certainly 0
   (2) if $i \ll \log_2(n)$, $BITMAP[i]$ is almost certainly 1

f) Key concept: The key idea of the approach is in using the value R, defined as the leftmost 0 in $BITMAP$, as an indicator of $\log_2(n)$

g) One can show that $\mathbb{E}(R) \simeq \log_2 \phi n$ with $\phi = 0.77351$ and $\sigma(R) \simeq 1.12$

5. Possible Adaptations 1
   a) Problem: The issue with the previous idea is that the dispersion of the results corresponds typically to one binary order of magnitude (factor of 2)
      (1) If we are off, we are at least off by factor of 2
   b) Alternative 1:
      (1) Repeat the estimation multiple times to decrease dispersion:
         (a) Use a set H of m hash functions and compute m different $BITMAP$ vectors, obtaining m estimates $R^{(1)}, R^{(2)}, \ldots, R^{(m)}$
         (b) Compute the arithmetic mean of all estimates:
$$A = \frac{\sum_{i=1}^{m} R^{(i)}}{m}$$
      (2) This approach reduces variability and stabilises the estimation leading to an improved performance:
$$\mathbb{E}(A) \simeq \log_2 \phi n \quad \sigma(A) \simeq 1.12/\sqrt{m}$$
$$m \uparrow \text{ dispersion } \downarrow$$

6. Possible Adaptations 2
   a) Problem: The previous adaptation increases running time by a factor m
   b) Alternative 2: Instead, we will use an approach called *stochastic averaging* for filling the individual arrays $BITMAP_i$
      (1) Use the hashing function to distribute each record into one of m possible slots:
$$\alpha = h(x) \,\%\, m$$
      (2) Update slot $BITMAP_\alpha[i]$ using $i = \rho(h(x)/m)$
      (3) compute arithmetic mean $A = \frac{\sum_{i=1}^{m} R^{(i)}}{m}$
   c) We expect that $\simeq \frac{n}{m}$ items fall into each slot and hence that $\frac{1}{\phi} 2^A$ is a reasonable estimate of $\frac{n}{m}$. This adaptation is called Probabilistic counting

with stochastic averaging (PCSA). One can prove a relative accuracy of $\frac{0.78}{\sqrt{m}}$

IV. Use cases for cardinality estimation
   A. How is this practically relevant?
      1. In bioinformatics streams of sequencing data are relatively common
      2. Example: Classification of whole meta genome sequencing reads
      3. Task:
         a) given a set (stream) of high-throughput sequencing reads form a sample containing a population of diverse microorganism in the order of 10^8 reads
         b) assign each read to a label from a given database
         c) compute the composition of label frequencies over the give sample.
      4. Idea: Use cardinality estimation to solve this problem. The software KrakenUniq employs HyperLogLog counters
   B. Link to Genome Comparison
      1. Similar to min-hashing, also HLL counters can be used to estimate similarity between string sets
      2. Based on HLL sketches one can count, integrate and compare different datasets.
      3. Using MLE for HLL estimation is slightly slower than MinHash (can be improved with SIMD) but shows drastically improved accuracy.
V. Example Questions
   A. What is cardinality of a multi-set
      1. The cardinality of a multi-set is the number of distinct elements in that multiset
   B. Can you describe the technique called stochastic averaging, used in Flajolet-Martin algorithm?
      1. This is an adaptation from Flajolet-Martin algorithm. Instead of 1 BITMAP, it uses m BITMAP.
      2. For each element x in the multiset M, we first hash the element and then mod it by m. $\alpha = hash(m) \,\%\, m$
      3. Then it updates the m-th BITMAP's i-th position using $i = \rho(hash(x)/m)$ where ρ is a function returning the left most 0
      4. Finally we compute the arithmetic mean $A = \frac{\sum_{i=1}^{m} R^{(i)}}{m}$ and calculate the cardinality estimation as $\frac{m}{\phi} 2^A$

Lecture 11: Secure algorithms for genomics

I. Motivation for privacy preserving computation
   A. Processing of genomics data comes with a number of privacy concerns:
      1. genetic data of human samples is inherently personal (it cannot be anonymised without losing relevant signal)
      2. data aggregation is inevitable (and necessary) to learn about relevant properties of a population or to infer function
      3. integration of multiple data types (and sources) is required for biomedical analyses (increasing the risk of identification rather than decreasing it)
   B. Solution: encrypt any communication containing confidential data
   C. Question How?

II. Key-exchange protocols
   A. One of the key suggestions was the Diffie-Hellman key-exchange protocol:
   B. Goal: Given a client A and a server B, establish a shared secret between them via a public channel.
   C. Approach:
      1. A and B agree on a joint starting value v
      2. A and B each have a personal secret s_A and s_B
      3. each party uses their personal secret to modify the joint value

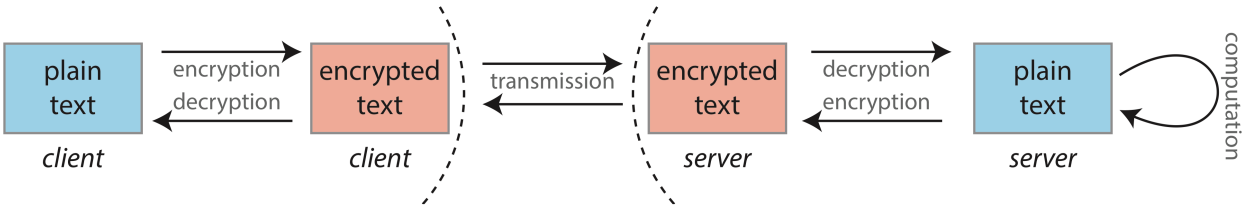
$$v \oplus s_A \rightarrow v_A \text{ and } v \oplus s_B \rightarrow v_B$$

      (assuming it is hard/impossible to infer s_A when v_A and v are given)
      4. parties exchange values v_A and v_B
      5. parties independently create their shared secret v_S as

$$v_B \oplus s_A \rightarrow v_S \text{ and } v_A \oplus s_B \rightarrow v_S$$

   D. Public key infrastructure
      1. An alternative (or additional security measure) is to associate each party partaking in the communication with a unique public key
         a) each party applies to “certificate authority”
         b) certificate identifies each user unambiguously
         c) messages are encrypted with a user’s public key (and only the user can decrypt with their private key)
      2. Considerations
         a) requires that a trustworthy authority exits to identify a user (centralisation)
         b) still, only communications is encrypted and either party needs to decrypt

III. Homomorphic encryption
   A. Encrypted communication
      1. The typical flow for an encrypted communication looks as follows:



      2. Problems:
         a) Server has full access to client’s query
         b) client can iteratively query and obtain all contents from server

c) compromised server creates vulnerability for plain client data

3. Question: How can secure communication via a public channel be achieved

B. Communication between untrusting parties

1. Communication though parties not trusting each other can have legal (or other motivations).
2. Scenario: Assume a request to a hospital's database containing the genetic variation of all patients
   a) A possible query could be: How many patients with variant combination XYZ do exist in the hospital's database?
3. Question: Can the query be answered without the hospital knowing which variant combination was queried and without client knowing which other variants are present in the hospital's database?

⇒ Motivation for **privacy-preserving computation**

C. Privacy-preserving computation

1. Idea: Perform computations on encrypted data (the cipher text) and only decrypt the result once the message has reached again the requesting entity
2. Concept: For this to work, it is necessary that alterations of the cipher text can be mapped back implicitly to the plain text and that such mapping happens in a way that the encryption is not weakened through said transformation

⇒ We will use **Homomorphic encryption** to achieve this

D. Homomorphic encryption

1. Homomorphic encryption is based on the mathematical concept of homomorphy
2. Homomorphism: A homomorphism is a map between two algebraic structures (e.g., groups) that preserves the operations of the structures, e.g.

$$f(a \circ b) = f(a) \bullet f(b)$$

3. Example: The map between the positive reals and their logarithm is a homomorphism:

$$a \cdot b = c \Rightarrow log(a) + log(b) = \log(c)$$
$$\log(a \cdot b) = \log(c)$$

   a) $\log(a \cdot b) = \log(a) + \log(b)$

E. Classes of homomorphic encryption

1. Depending on the number and depth of operations allowed, we can distinguish different classes of homomorphic encryption:
   a) **PHE** (partially homomorphic encryption): only either addition or multiplication are supported, but not both
   b) **SWHE** (Somewhat homomorphic encryption): both addition and multiplication are possible, but only up to a limited number of operations
   c) **FHE** (fully homomorphic encryption): both addition and multiplication for an unlimited number of operations

⇒ For many current practical implementations, a non-fully homomorphic encryption system is sufficient

F. Practical use case

1. A possible use-case is beacon-type queries for genomic variation
2. A suggested secure algorithm uses PHE based on single addition operations only for privacy preserving string matching

IV.