Compressed text indexing

- FM-index: compressed Burrows Wheeler Transform (BWT)   [Ferragina, Manzinni, focs 00]

- Compressed suffix arrays: [Grossi & Vitter, stoc 00]

Space: O(n) bits (instead of O(n) words)

Today's lecture

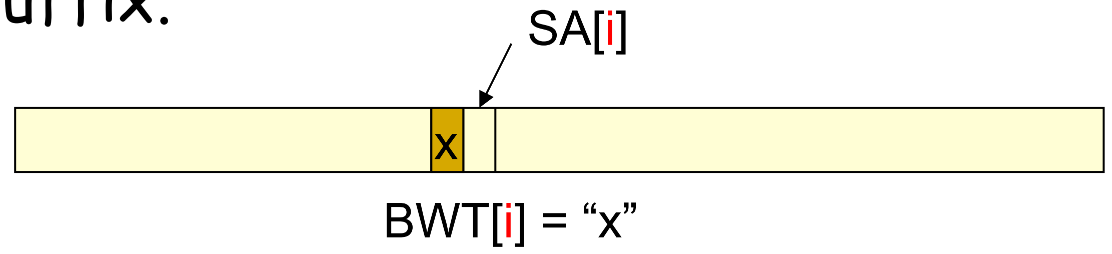
- BWT for pattern matching
- 2n bits for DNA; i.e, same size as the text

Definition of BWT [Burrows & Wheeler 94]

- T[0..n] -- a text of n+1 characters, where T[n] = $ (a special character).
- SA[0..n] -- SA[i] = j if T[j..n] is the ith lexicographically smallest suffix.

  The rank of T[j..n] is said to be i.
- BWT[i] = T[j-1], where j = SA[i].

  I.e., BWT[i] is the character immediately before the ith smallest suffix.



Example

- T = ACACGT$
- BWT = T$CAACG

- BWT[i] is the character immediately before the i^th suffix.

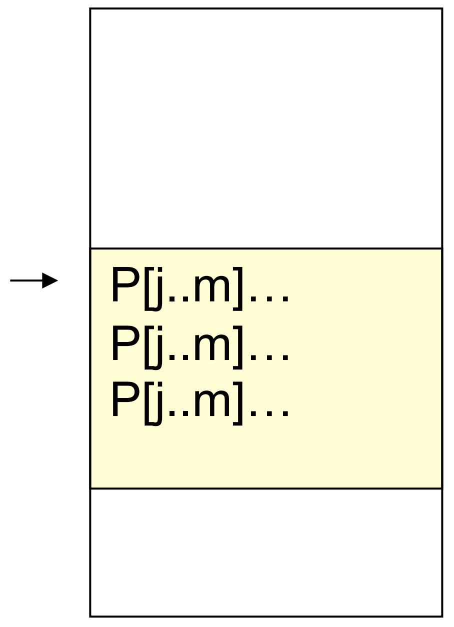
	SA	BWT	Sorted Suffices
0	6	T	$
1	0	$	ACACGT$
2	2	C	ACGT$
3	1	A	CACGT$
4	3	A	CGT$
5	4	C	GT$
6	5	G	T$

Backward searching

BWT (plus some auxiliary function) allows us to search for a pattern in a backward manner.

Given a pattern P[1..m], compute

- the (smallest) rank of P[m] w.r.t. the suffix array of T
- the (smallest) rank of p[m-1..m]
- …
- the (smallest) rank of p[2..m]
- the (smallest) rank of p[1..m]



NB. The largest rank is computed in a similar way.

Count(x)

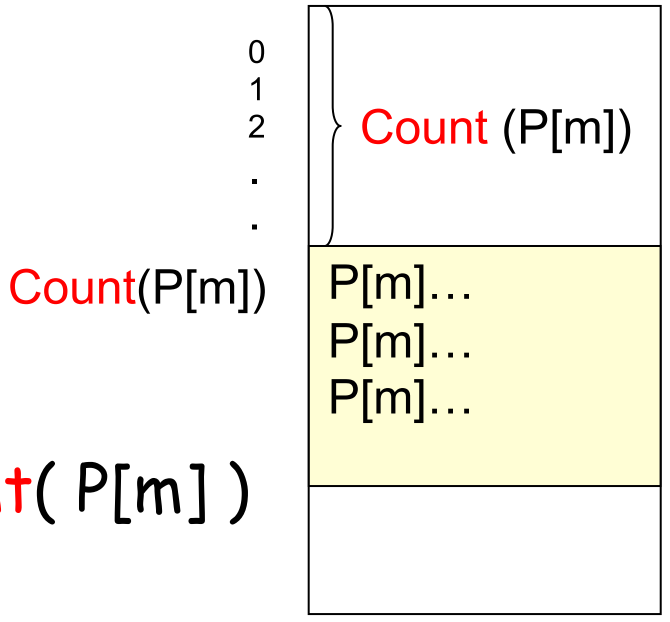
- For any character x, let Count(x) be the total # of characters in the text T that are smaller than x.

  E.g., if T = ACACGT$, then

  Count(A) = 1;

  Count(C) = 3;

  Count(G) = 5



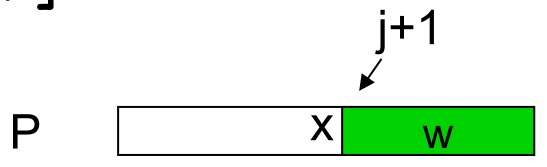
- (smallest) Rank of P[m..m] = Count(P[m])

  E.g., P = ACG; Count(G) = 5

From P[j+1..m] to P[j..m]

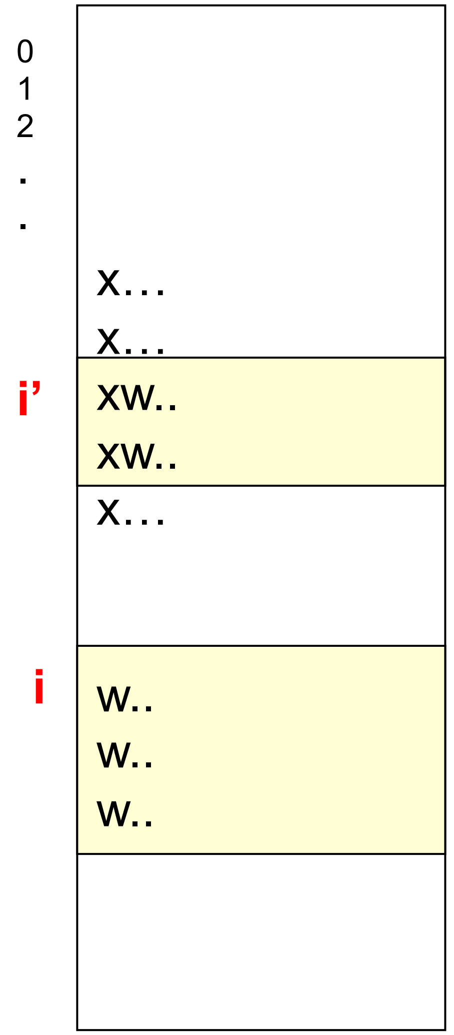
Let P[j+1..m] = w, and let P[j] = x.

Then P[j..m] = xw.



Suppose we've already computed the (smallest) rank of w, which is equal to i.

Next, we want to compute the (smallest) rank of xw (denoted i' in the figure).



Computing rank of xw

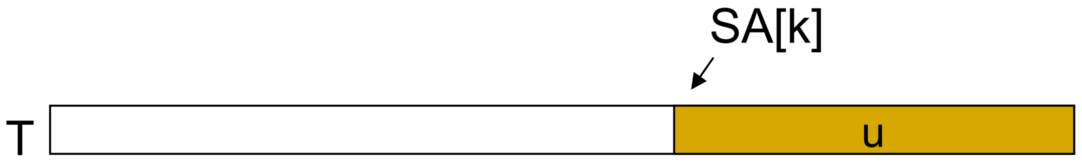
Rank of xw = Count(x) +

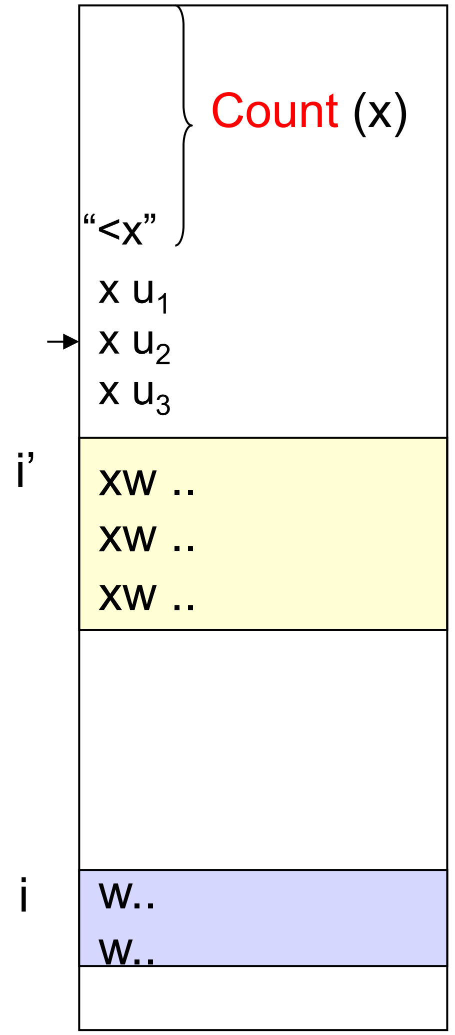
# of suffixes xu such that u $<_\ell$ w

Consider a suffix xu with u $<_\ell$ w.

- u $<_\ell$ w ⇔ rank of u < (smallest) rank of w.
- Recall that (smallest) rank of w = i.

  Let rank of u = k.   Then k < i.





Computing rank of xw

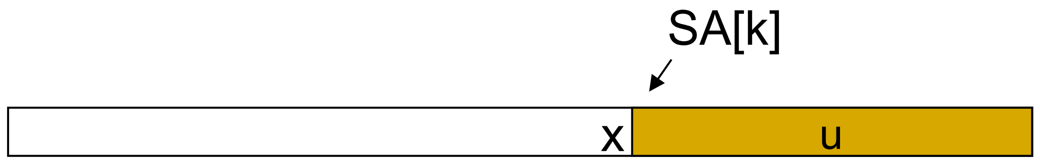
Rank of xw = Count(x) +

  # of suffixes xu such that u < w

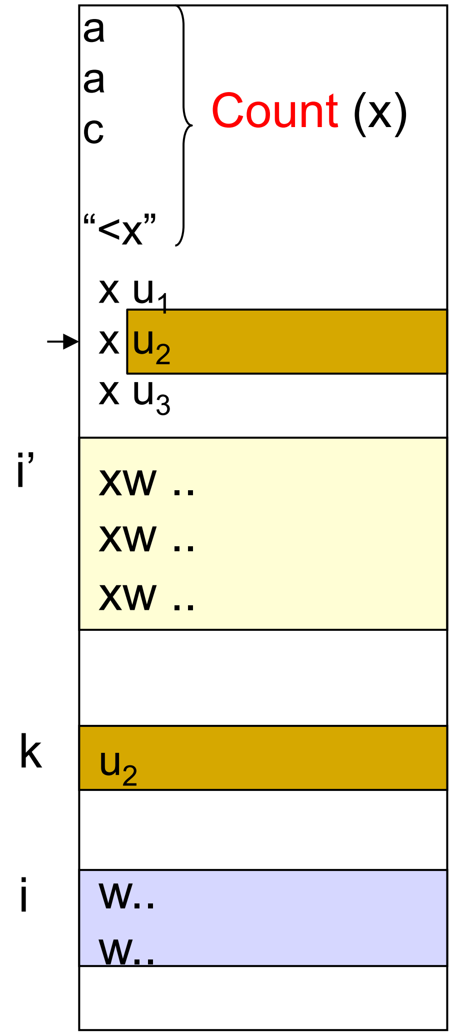
Consider a suffix xu of T with u < w.

- u < w ⇔ rank of u < (smallest) rank of w.
- Recall that (smallest) rank of w = i.

  Let rank of u = k. Then k < i.



- Then k < i and BWT(k) = "x"



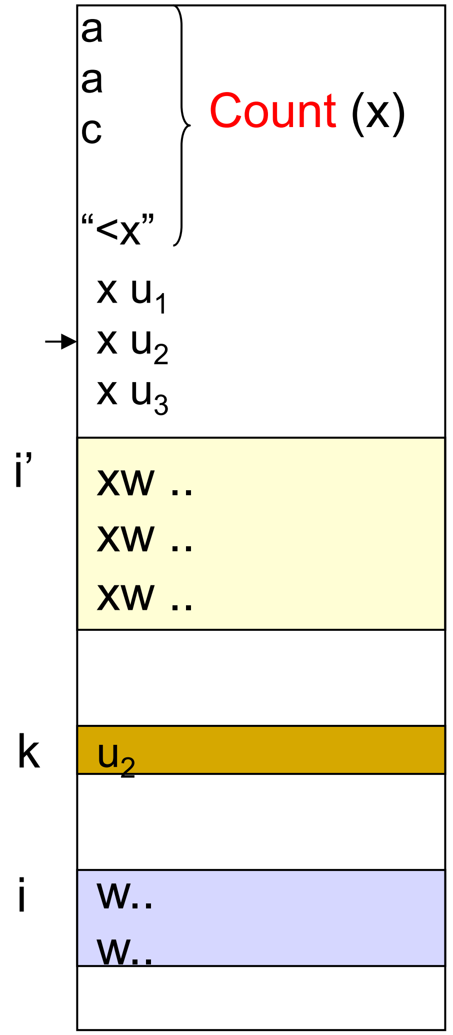
Use BWT to compute rank of xw

Rank of xw = Count(x) +

# of suffixes xu such that $u <_\ell w$

Each suffix xu with $u <_\ell w$
corresponds uniquely to
an index k < i such that BWT(k) = "x".

  # of suffixes xu of T such that u < w
= # of indices k < i such that BWT(k) = "x"
= # of x's in BWT[0..i-1]



Example

- T = ACACGT$
- P = ACG
- Notation: **Appear**[i,x] = # of x in BWT[0..i-1]
- Rank of G = Count(G) = 5

	SA	BWT	Sorted Suffices
0	6	T	$
1	0	$	ACACGT$
2	2	C	ACGT$
3	1	A	CACGT$
4	3	A	CGT$
5	4	C	GT$
6	5	G	T$

Example

- T = ACACGT$
- P = ACG
- Notation: Appear[i,x] = # of x in BWT[0..i-1]
- Rank of G = Count(G) = 5
- Rank of CG = Count(C) + Appear(5,C) = 3 + 1 = 4

	SA	BWT	Sorted Suffices
0	6	T	$
1	0	$	ACACGT$
2	2	C	ACGT$
3	1	A	CACGT$
4	3	A	CGT$
5	4	C	GT$
6	5	G	T$

Example

- T = ACACGT$
- P = ACG
- Notation: Appear[i,x] = # of x in BWT[0..i-1]

Rank of G = Count(G) = 5

Rank of CG = Count(C) + Appear(5,C) = 3 + 1 = 4

Rank of ACG = Count(A) + Appear(4,A) = 1 + 1 = 2

	SA	BWT	Sorted Suffices
0	6	T	$
1	0	$	ACACGT$
2	2	C	ACGT$
3	1	A	CACGT$
4	3	A	CGT$
5	4	C	GT$
6	5	G	T$

Searching details

For any pattern P[1..m], for any $1 \le j \le m$, let

- $first_j$ be the SA index (rank) of the first suffix matching P[j..m]; and
- $last_j$ be the SA index (rank) of the last suffix matching with P[j..m].
  - $first_m$ = Count(P[m])
  - $last_m$ = Count(P[m] + 1) - 1
  - ...
  - $first_j$ = Count(P[j]) + Appear(P[j], $first_{j+1}$)
  - $last_j$ = Count(P[j]) + Appear(P[j], $last_{j+1}$ + 1)

Preprocessing

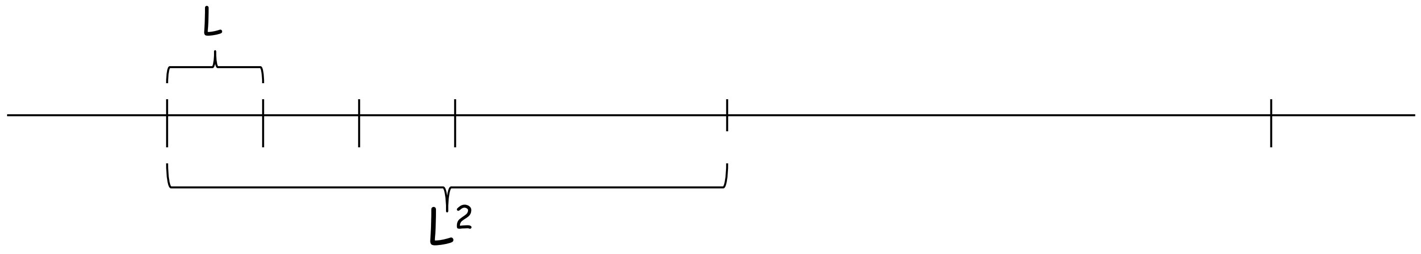
- To save time, we perform preprocessing on Appear & Count.

  $o(|\Sigma|n)$ bits are sufficient to allow very efficient retrieval of Count and Count.

- FM-index is a compressed version of BWT (e.g. using move-to-front and run-length encoding). However, for DNA ($|\Sigma| = 4$), the compression is not effective and doesn't save much space.
- Finding the actual occurrences requires additional auxiliary data structures (SA sampling).

Appear: simple implementation

- BWT itself: n log |Σ| bits
- Auxiliary data structure for computing Appear: o(n|Σ|) bits
- Take L = O(log n), for every L^2 BWT characters, we store the Appear() value.
- For every L BWT character, we store the difference w.r.t. the last L^2 BWT position. This takes only $\log(L^2)$ bits as the value must be within L^2.



- In total, this takes $|\Sigma| (n / L^2 (\log n) + n / L (\log L^2)) = O(|\Sigma| (n / \log n + n \log \log n / \log n)) = o(n|\Sigma|)$ bits.

BWT segments of log n characters

- Given a segment of log n characters and a character x, we want to find how many times x appears in the segment using O(1) time.
- Each character is represented by log Σ bits.
- Consider a short segment of h=10 characters.
  - Index: In advance count the number of x in every possible segment of h characters and store the counts in a table (array) A indexed by "h characters" (i.e., a binary sequence of h log Σ bits).
  - Counting: Given any h characters, counting takes O(1) time.
- Let $h = \sim \frac{1}{2} \log n / \log \Sigma$.

  For each character x, build a table A_x

  Table A_x : $2^{\frac{1}{2} \log n} = n^{\frac{1}{2}}$ entries

Alternative: Rank & Select

- For each character x, define a bit vector V such V[i] = 1 if and only if BWT[i] = x.
- Build a rank-and-select data structure for V, then Appear[x,i] can be computed using a rank operation.
- Space: for each character, o(n) bits (precisely, (1 + o(1)) n log log n/log n + O(n/log n) bits).

Space-efficient Construction of BWT

- Don't build SA first. Instead build BWT direct.
- [Hon et al focs 03 & Algorithmica 07, Lippert et al. JCB 05]
- Requires only 2.5 n bits of memory; takes 20 minutes to build a BWT for the human genome (on an ordinary PC).
- Converting BWT to FM-index or CSA is even faster.

- For DNA, BWT (uncompressed FM-index) has the best performance regarding space & time.

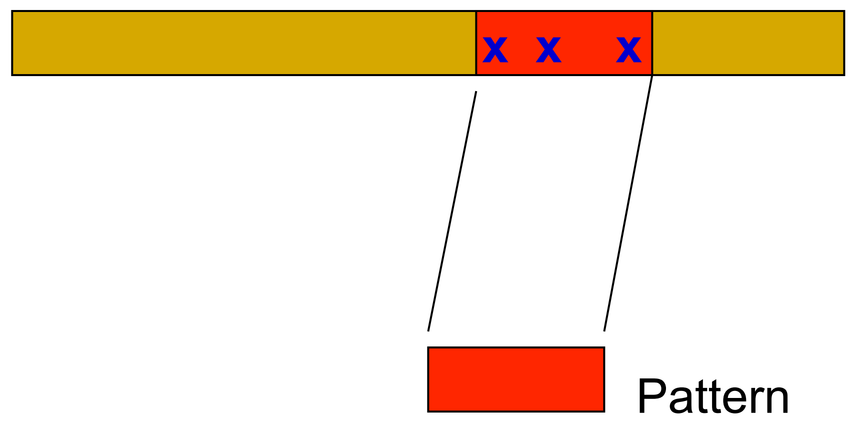
Survey paper

- Compressed full-text indexes, Navarro & Makinen, ACM Computing Surveys, April 2007

Text Indexing

Build an index for a DNA sequence T (say, human genome) in the main memory to facilitate

- Exact pattern matching
- Approximate string matching (semi-global alignment)



Approximate string matching

Not trivial even when the number of errors, k, is small.

- Suffix tree: $O(n)$ words, $O(|P|^k + occ)$ time
- $O(n \log^k n)$-word index: $O(|P| + \log^k n \log\log n + occ)$ time [Cole et al. STOC 04]
- $O(n \log^{k-1} n)$-word index, same k-error matching time [Chan et al. ESA 06]
- $O(n)$-word index: $O(|P| + \log^{k(k+1)} n \log\log n + occ)$ time [Chan et al. CPM 06]

These indexes occupy more space than a suffix tree. Too big for indexing long DNA sequences.