Introduction to Bioinformatics

Transcription factor binding site prediction

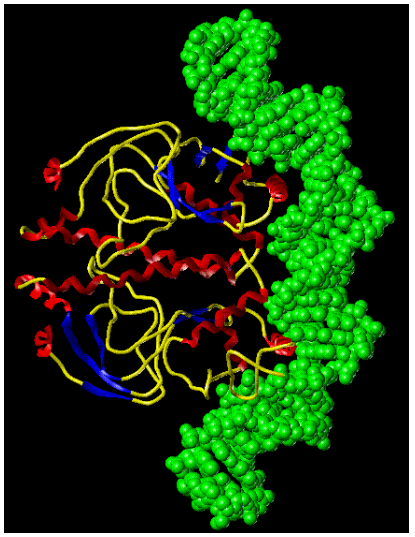
Lecturer: Jan Baumbach
Teaching assistant(s): Diogo Marinho

Outline

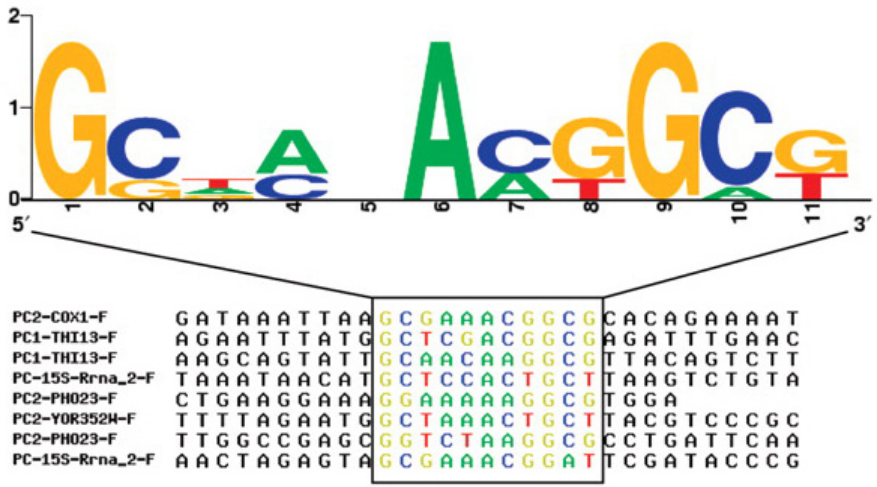
- Method without considering background distribution
- General approach considering background distribution
- Ways to speed up the algorithm

Transcription Factor Binding Sites (TFBSs)

Transcription Factor Binding Sites (TFBSs)

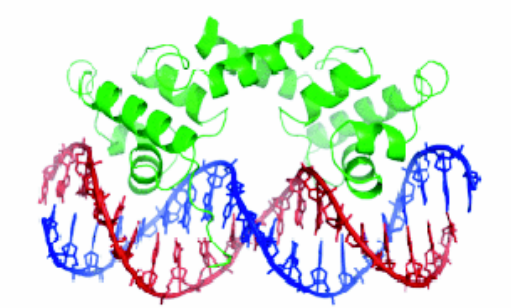


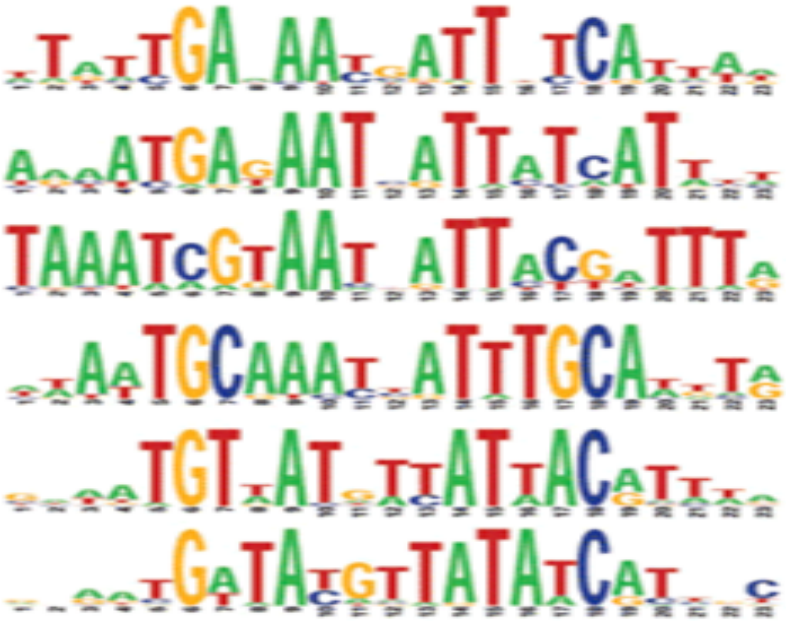
- DNA sequence segments that transcription factors (TF) bind to are called transcription factor-binding sites (TFBSs).
- TF interact with their TFBS using a combination of electrostatic and Van der Waals forces.
- Most of the TFs bind DNA in a motif specific manner, i.e. TFs can bind to a list of similar DNA sequence segments.



Transcription Factor Binding Sites (TFBSs)

- Transcription factor binding sites are usually short (around 5-15 bp)
- They are frequently degenerate sequence motifs
  - The sequence degeneracy confers different levels of regulation
- Given a genome, the prediction of TFBSs is a difficult and risky task.





Identification of TFBSs

- Experiment methods
  - Traditional methods
    - Foot-printing methods
    - Nitrocellulose binding assays
    - Gel-shift analysis
    - Southwestern blotting
  - High-throughput method
    - Finding high-affinity binding sequence *in vitro* (SELEX)
    - High-throughput method *in vivo*: ChIP-chip
- TFBSs *in silico*
  - Aim: to identify more candidate target TFBS.
  - Degenerate consensus sequences. (Drawback: does not contain precise likelihood information)
  - Position weighted matrix (PWM) or PSSM (position specific matrix) is a common approach to this problem.

Position Weight Matrices (PWMs)

Position weight matrix (PWM)/ Position-specific weight matrix(PSSM)

PWM is a commonly used representation of motifs(patterns) in biological sequences.

Imagine two experimentally determined TF binding sites for one TF:

Seq1: ATTGAGTCGCAGTGACTCAAG

Seq2: CTTGAGTCAGGCAGGCTCAAT

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	1	0	0	0	2	0	0	0	1	0	1	0	1	0	1	0	0	0	2	2	0
T	0	2	2	0	0	0	2	0	0	0	0	0	1	0	0	0	2	0	0	0	1
C	1	0	0	0	0	0	0	2	0	1	0	1	0	0	0	2	0	2	0	0	0
G	0	0	0	2	0	2	0	0	1	1	1	1	0	2	1	0	0	0	0	0	1

PWM of "better quality":

Constructed using 33 TF binding sites for one TF

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	3	8	26	5	0	5	10	28	5	0	3	4	22	20	4	2	1	23	19
T	26	22	2	3	1	20	18	4	1	1	7	28	6	2	2	6	29	9	7
C	1	2	3	2	0	6	1	0	3	11	20	1	5	2	26	25	1	0	3
G	3	1	2	23	32	2	4	1	24	21	3	0	0	9	1	0	2	1	4

Definitions:

- **Length** of PWM (number of columns): M
  - **absolute PWM (count matrix)**:

$f_{i,j}^{*} \in N$ with $i \in \{A,T,C,G\}$ and $j \in [0, M-1]$

  - **relative PWM (frequency matrix)**:

$$f_{i,j}' = \frac{f_{i,j}^{*}}{\sum_{k \in \{A,T,C,G\}} f_{k,j}^{*}}$$

	1	2	3
A	0.8	0	..
T	0.1	0	..
C	0.1	0.2	..
G	0	0.8	..

  - **Pseudo counts** per column (avoid overfitting): e.g. $c = 4$

$$f_{i,j}^{p} = f_{i,j}^{*} + c\pi_i \quad \rightarrow \quad f_{i,j} = \frac{f_{i,j}^{p}}{\sum_{k \in \{A,T,C,G\}} f_{k,j}^{p}}$$

A simple TFBS matching tool

Naïve method without considering background distribution

MATCH™: a tool for searching transcription factor binding sites in DNA sequences (A.E. Kel *et al.* 2003)

- Input:
  (1) DNA sequences containing potential TF binding sites
  (2) PWM

  Output:

  A list of found potential sites.

- Two types of scores are calculated
  - Core Similarity Score (CSS) : only calculated for the first five consecutive conserved region.
  - Matrix Similarity Score (MSS): calculated for all the positions

Naïve method without considering background distribution

MATCH™: a tool for searching transcription factor binding sites in DNA sequences (A.E. Kel *et al.* 2003)

$$MSS(CSS) = \frac{Current - Min}{Max - Min} \qquad MSS(CSS) \in [0,1]$$

$nuc(j)$ refers to the nucleotide with index j.

$$Current = \sum_{j=0}^{L-1} I(j) f_{nuc(j),j}' \qquad f_{i,j}' = f_{i,j}^* \Big/ \sum_{k \in \{A,T,C,G\}} f_{k,j}^*$$

$$Max = \sum_{j=0}^{L-1} I(j) f_j^{\max} \qquad f_j^{\max} = \max_i \{f_{i,j}^*\} \Big/ \sum_{k \in \{A,T,C,G\}} f_{k,j}^*$$

(highest frequency of nucleotide in position j in the matrix)

$$Min: \sum_{i=0}^{L-1} I(j) f_j^{\min} \qquad f_j^{\min} = \min_i \{f_{i,j}^*\} \Big/ \sum_{k \in \{A,T,C,G\}} f_{k,j}^*$$

(lowest frequency of nucleotide in position j in the matrix)

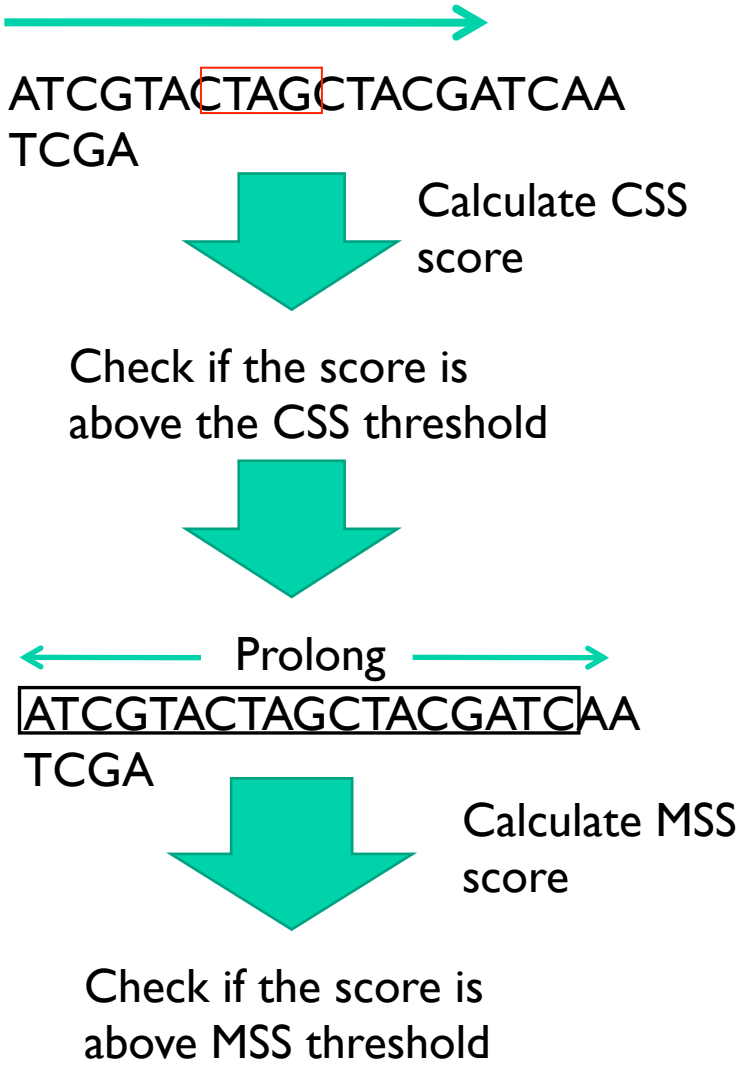
$$I(j) = \sum_{i \in \{A,T,G,C\}} f_{i,j} \ln(4 f_{i,j}) \qquad j = 1,2,\ldots,L$$

Information vector

- Two cutoffs are kept for CSS and MSS scores respectively.

Procedure:

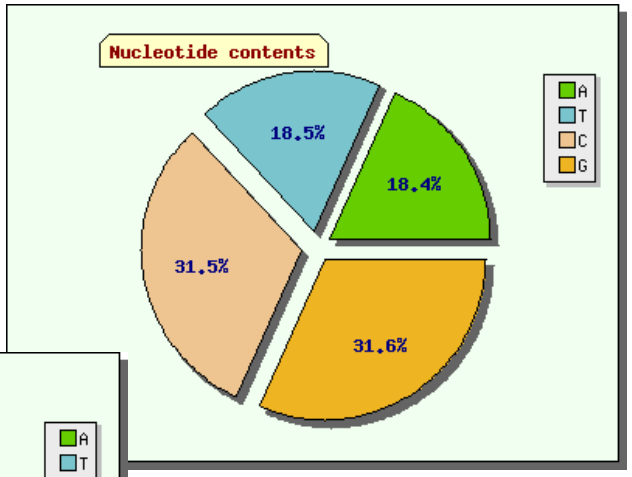
- A window consisting of five nucleotides is moving along the sequence.
- CSS (core similarity score) is calculated.
- For each CSS higher than CSS cut-off, the sequence and is prolonged at both ends to fit the matrix length. Then the MSS score is calculated
- If two scores are both higher than cut-offs, then output as a "yes" instance



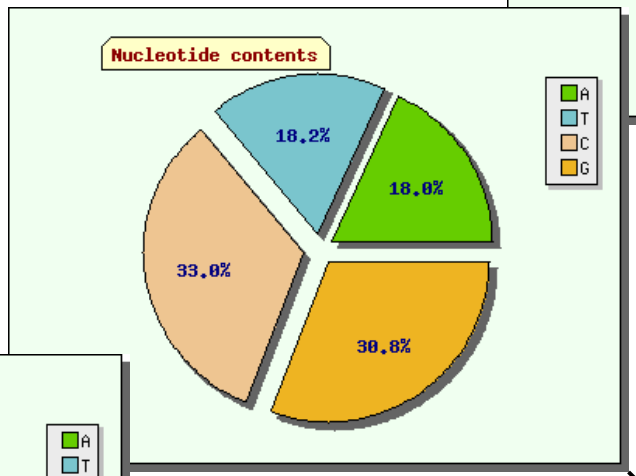
Incorporating the background

Background model:

- Some nucleotides in the PWM count more than others

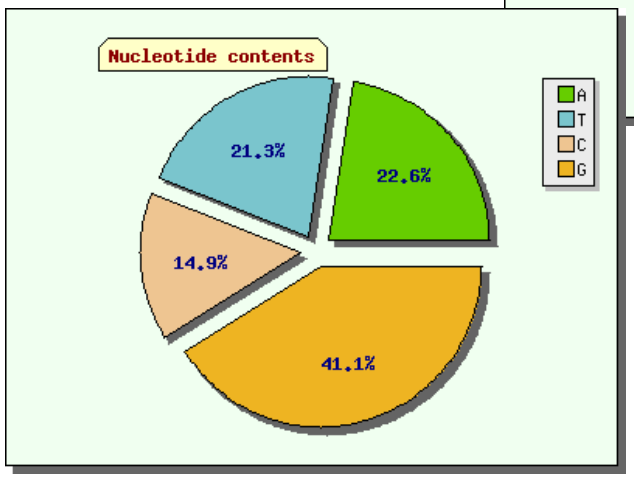


Nucleotide contents (total), C. efficiens



Nucleotide contents (coding), C. efficiens

Nucleotide contents (noncoding), C. efficiens



	relative.	absolute
A	18 %	520789
T	18.2 %	525858
C	33 %	952830
G	30.8 %	889610

Definitions:

- **Length** of PWM (number of columns): M
  - **Background model**:

$\pi_i \in [0,1]$ with $i \in \{A,T,C,G\}$ with $\sum_i \pi_i = 1$

A	0.180
T	0.182
C	0.330
G	0.308

  - **absolute PWM (count matrix)**:

$f_{i,j}^{*} \in N$ with $i \in \{A,T,C,G\}$ and $j \in [0, M-1]$

  - **relative PWM (frequency matrix)**:

$$f_{i,j}' = \frac{f_{i,j}^{*}}{\sum_{k \in \{A,T,C,G\}} f_{k,j}^{*}}$$

	1	2	3
A	0.8	0	..
T	0.1	0	..
C	0.1	0.2	..
G	0	0.8	..

  - **Pseudo-counts** per column (avoid overfitting): e.g. $c = 4$

$\rightarrow f_{i,j}^{p} = f_{i,j}^{*} + c\pi_i \quad \rightarrow \quad f_{i,j} = \frac{f_{i,j}^{p}}{\sum_{k \in \{A,T,C,G\}} f_{k,j}^{p}}$

Definitions:

- **Scoring function** (log-odds score):

$$S_{startIdx,endIdx} = \sum_{j=startIdx}^{endIdx} \ln \frac{f_{nuc(j),j}}{\pi_{nuc(j)}}$$

where $nuc(j)$ = nucleotide with index j

Matching procedure:

Seq = A G C A A T T A A A T T G G A T A A C..

PWM =

	1	2	3	4	5	6	7	8	9	10	11	12	13
A	19	17	16	17	1	16	14	15	3	2	18	19	18
T	13	18	20	3	2	20	20	21	1	12	15	12	11
C	3	1	0	2	31	1	1	1	3	22	3	2	4
G	2	1	1	15	3	0	2	0	30	1	1	3	3

- Calulate score $S_{0,M-1}$ for every position of the sliding window
- Report every match with $S_{0,M-1} > th$ (th is the threshold of being a signal)

But how to set a good threshold value?

Score distribution:

- $\lambda_B(X)$ score distribution of the PWM calculated with random sequences according to background model.

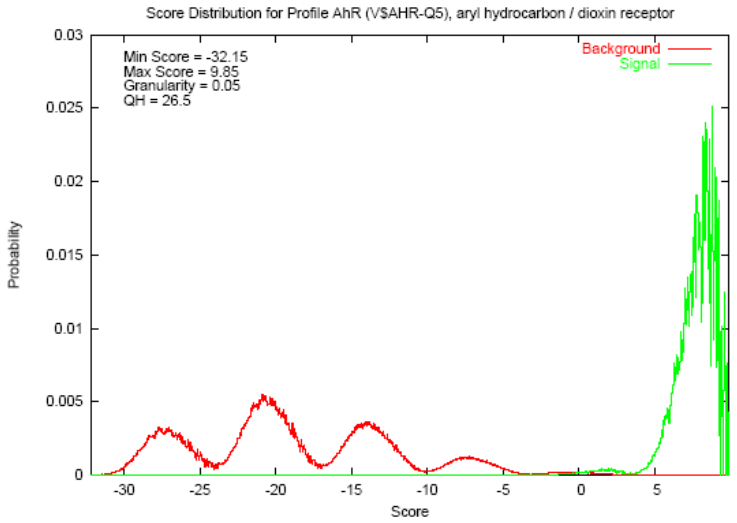
- $\lambda_T(X)$ score distribution calculated with rand sequences according to PWM model.

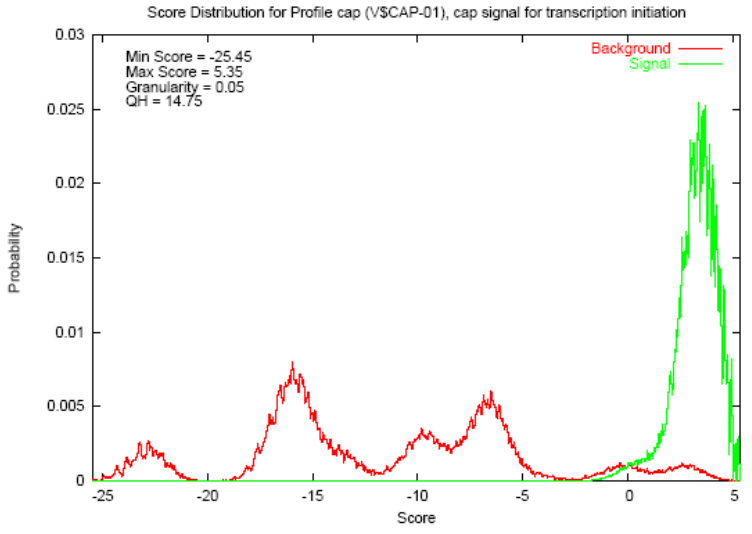
- $P_Z(X = s)$ probability of observing score s under distribution Z.

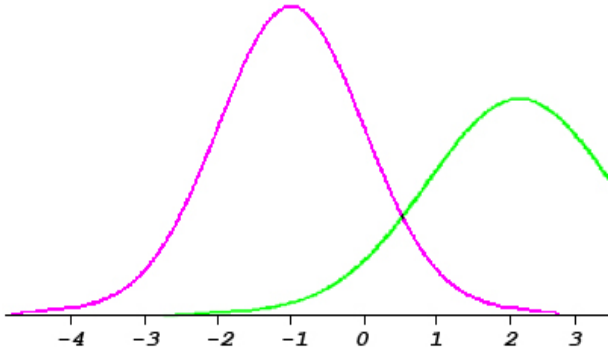
We are interested in:

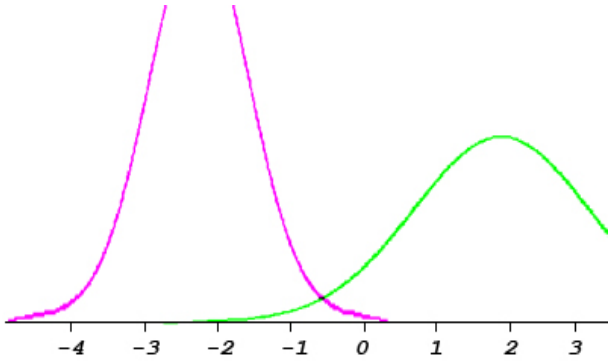
- $P_Z(X \geq s)$ probability of observing at least score s.

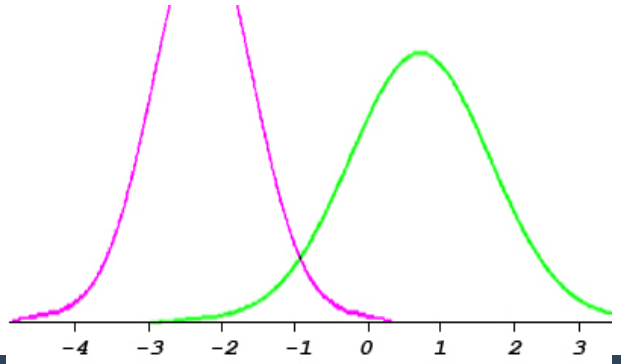
with $P_Z(X \geq s) = \sum_{i=s}^{\max} P_Z(X = i)$











pValue: $p = P_B(X \geq s) = \sum_{x=s}^{\max} P_B(X = x)$

Probability of observing at least score s by chance

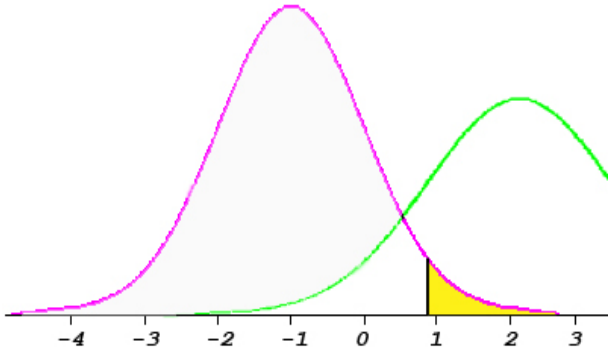
→ Set $th = s$, with $P_B(X \geq s) = p$

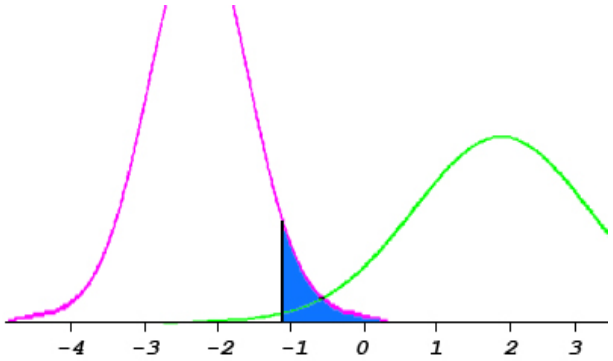
for given p

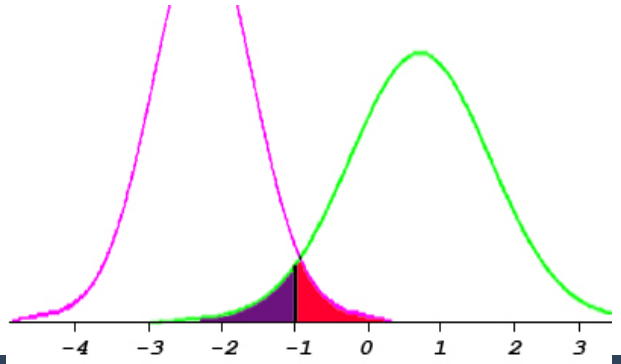
(pictures: assuming standard normal distribution)

Set equal false positive and false negative errors:

- Set s, where $P_B(X \geq s) = P_T(X \leq s)$







pValue: $p = P_B(X \geq s) = \sum_{x=s}^{\max} P_B(X = x)$

Probability of observing at least score s by chance

→ Set $th = s$, with $P_B(X \geq s) = p$

for given p

(pictures: assuming standard normal distribution)

Set equal false positive and false negative errors:

- Set s, where $P_B(X \geq s) = P_T(X \leq s)$

Methods to speed up general matching approach

- The general matching approach aims for finding a binding site by moving the window of length M along a sequence of length N.
- The time complexity of a straight-forward implementation is $O(MN)$
- Several methods were implemented to speed up the PWM/PSSM
  - Lookahead algorithm
  - Permutated lookahead algorithm
  - Suffix tree
  - Enhanced suffix array

Let's speed it up!

Kirk: "How much time to you need, Scotty?"
Scotty: "Gimme 20 minutes."
Kirk: "You got 10."
Scotty: "OK. I'll do in in 5."

… Two minutes later …

Lookahead algorithm

- **The motivation**: given a segment of sequence, we want to know whether we can reject its probability of being a signal as early as possible.

  - For a given sequence segment of length M, we have the score function:

$$S_{0,M-1} = \sum_{j=0}^{M-1} \ln(f_{nuc(j),j} / \pi_{nuc(j)}) \qquad \text{---(1)}$$

  - We define the minimum and maximum score for a given PWM:

$$S_{\min(0,M-1)} = \sum_{j=0}^{M-1} \min_{a \in \{A,T,G,C\}} \{\ln(f_{a,j} / \pi_a)\} \qquad \text{---(2)}$$

$$S_{\max(0,M-1)} = \sum_{j=0}^{M-1} \max_{a \in \{A,T,G,C\}} \{\ln(f_{a,j} / \pi_a)\} \qquad \text{---(3)}$$

Lookahead algorithm

- For any $0 \le d \le M-1$, we also define *the prefix score of depth d*:

$$pfxS_d = S_{0,d} = \sum_{j=0}^{d} \ln(f_{nuc(j),j} / \pi_{nuc(j)}) \qquad \text{---(4)}$$

- And the maximal score in the last M-d -1 positions of the PWM:

$$\sigma_d = S_{\max(d+1,M-1)} = \sum_{j=d+1}^{M-1} \max_{a \in \{A,T,G,C\}} \{\ln(f_{a,j} / \pi_a)\} \qquad \text{---(5)}$$

- Finally, we can calculate the *intermediate threshold* at a position *d:*

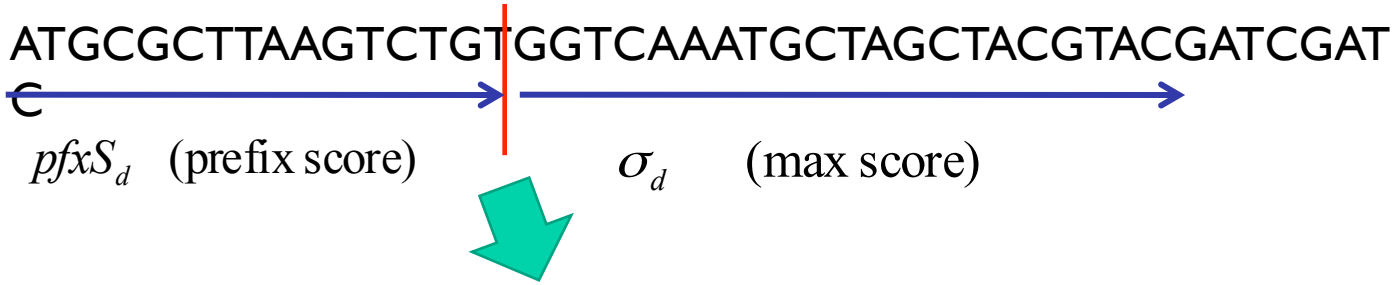
$$th_d = th - \sigma_d \qquad \text{----(6)}$$

Lookahead algorithm

- Therefore, the following statements are equivalent:

$$pfxS_d \geq th_d \text{ for all } d(0 \leq d \leq M-1)$$
$$\Leftrightarrow$$
$$S_{0,M-1} \geq th$$

- Basically, when a prefix has a score so low that even if the rest of the segment achieves maximal score, still the score for whole segment is below the threshold, then we must reject it.



Permutated lookahead algorithm

- With the lookahead algorithm, the sooner we reject a segment, the better running time we have.

- Therefore, it makes sense to check the positions in a PWM that is more likely to be rejected by lookahead algorithm. We implement this idea by a permutation of PWM:

- Each column of a PWM has a highest score:

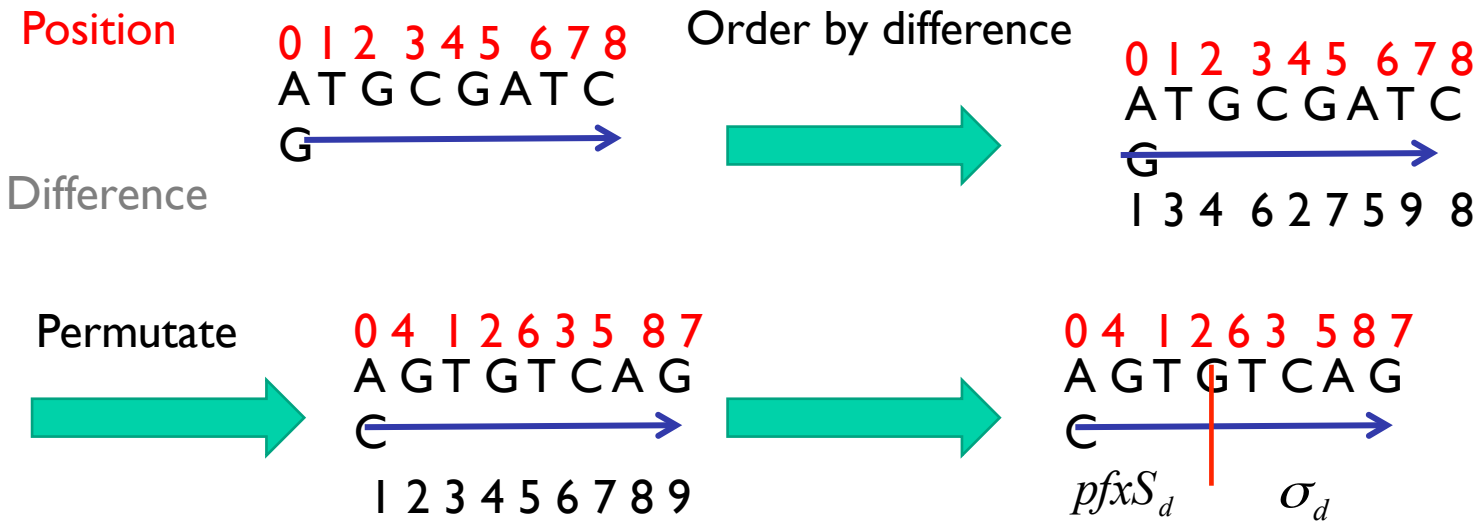
$$M_j = S_{\max(j,j)} = \max_{a \in \{A,T,G,C\}} \{\ln(f_{a,j} / \pi_a)\}$$

- and an expectation of the score if the residue is generated by background model:

$$E_j = \sum_{a \in \{A,T,G,C\}} S_{a,j} \cdot \pi_a = \sum_{a \in \{A,T,G,C\}} \ln(f_{a,j} / \pi_a) \cdot \pi_a$$

Permutated lookahead algorithm

- We focus on the difference between M_j and E_j. If the expectation for a column is comparatively low to the highest score, then it is more likely the segment is rejected at this column.
- Therefore, we order the matrix by $(M_j - E_j)$, and compute the most "dangerous" column first.

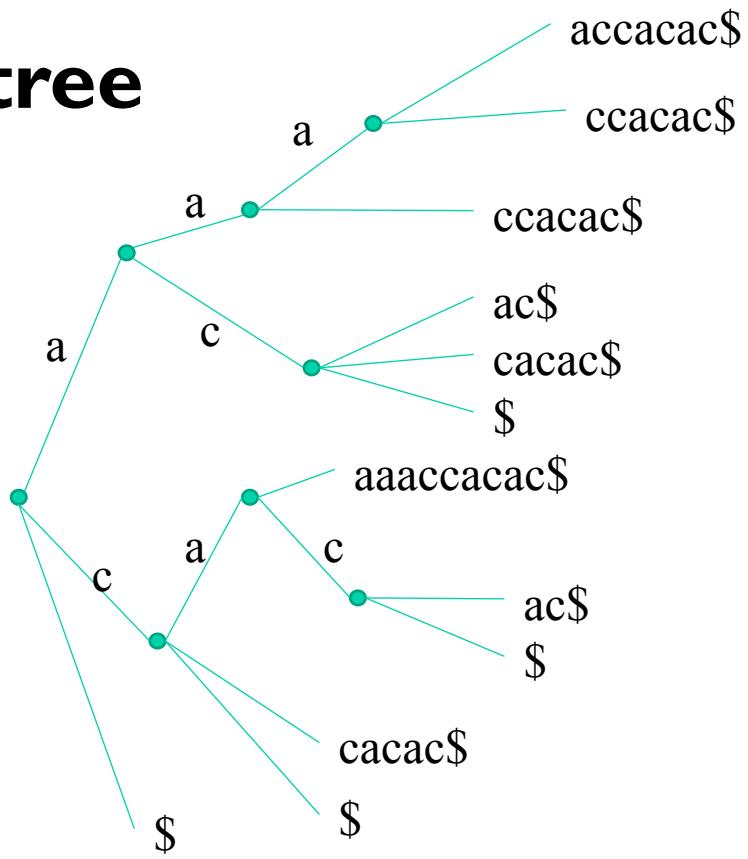


Suffix tree

1. Suffix tree is a data structure that presents all the suffixes of a given string.

2. A suffix tree for a string *w*, is a tree whose edges are labeled with substrings. Each suffix of *w* corresponds to exactly one path from the tree's root to leaf.

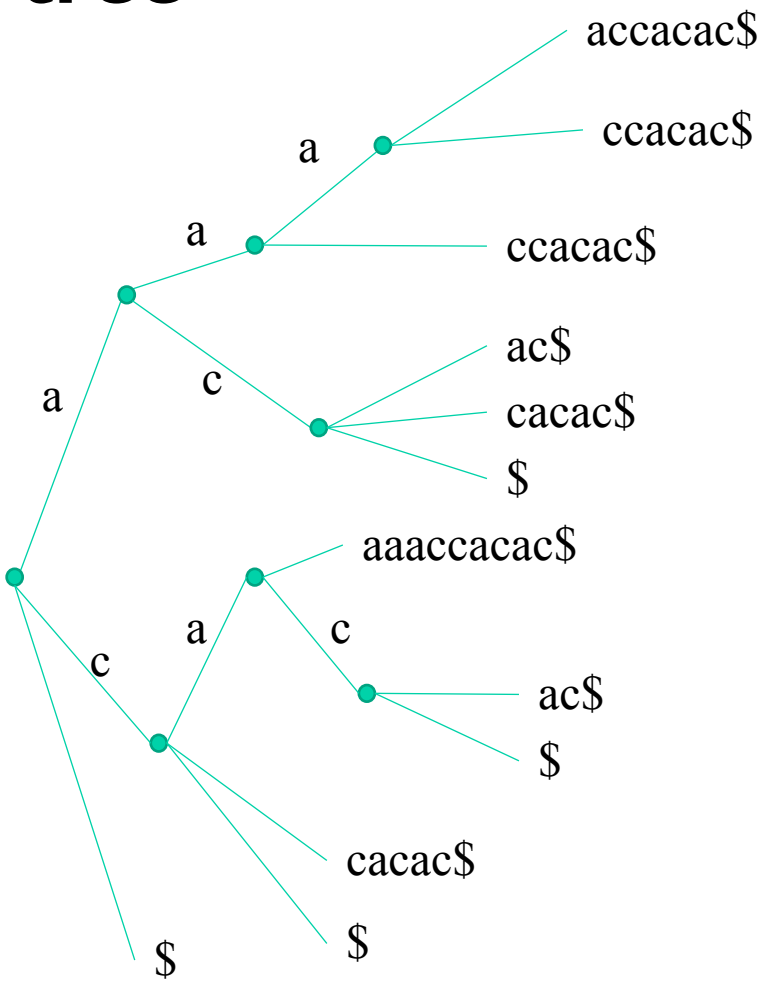
3. Suffix tree is a special data structure that allows a number of string operations to be carried out in an efficient way



Suffix tree for the string "caaaaccacac". Substring terminates with "$". The 12 paths from the root to a leaf correspond to the 12 suffixes.

Suffix tree

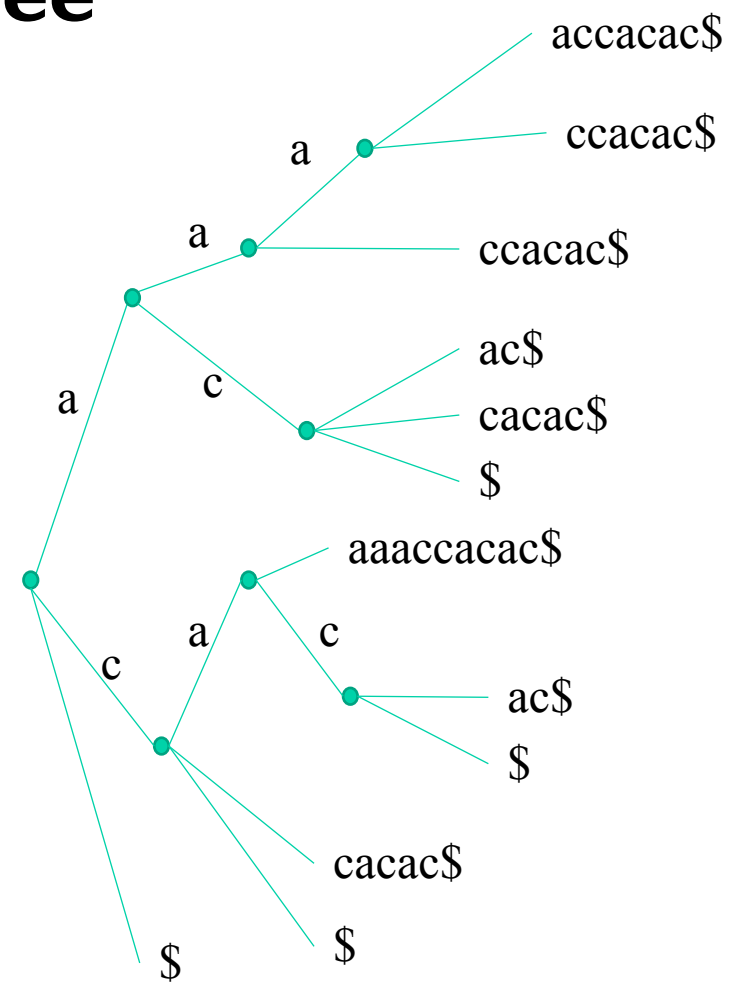
Number	Substring
0	aaaaccacac$
1	aaaccacac$
2	aaccacac$
3	acac$
4	accacac$
5	ac$
6	caaaaccacac$
7	cacac$
8	cac$
9	ccacac$
10	c$
11	$



Suffix tree

Key features of a **suffix tree** T for string w[0,…m-1] is a rooted tree with :

1. m leaves numbered from 0 to m-1
2. At least two children for each internal node (except root)
3. Each label represents a substring of w (nonempty)
4. No two edges out of the same node begin with same character

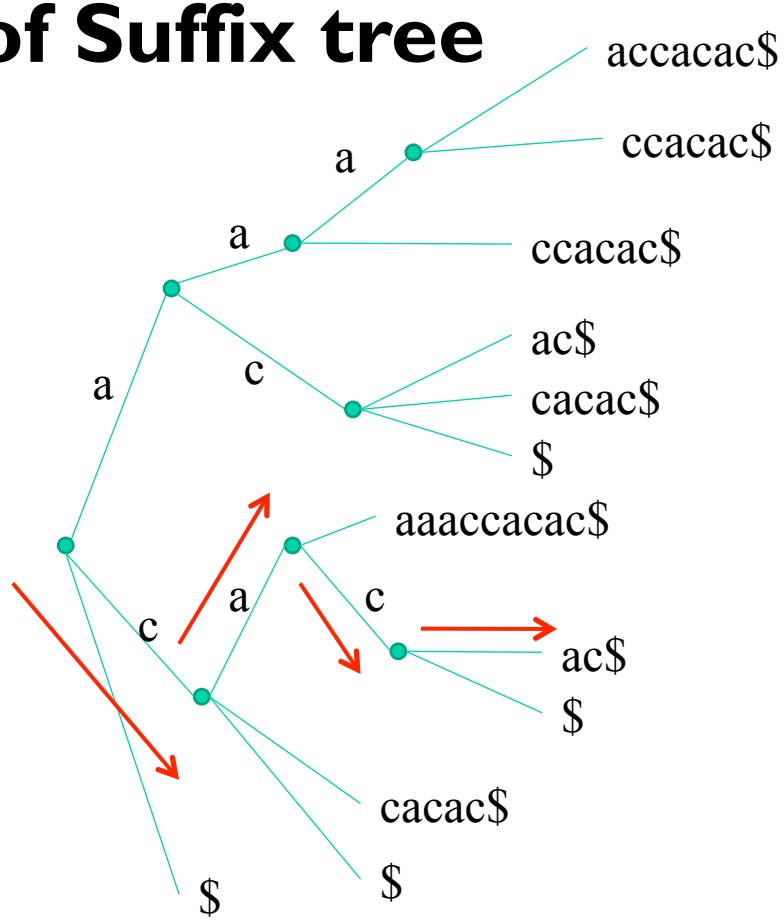


Applications of Suffix tree

- One of the simplest application of suffix tree is to check whether a string P of length m is a substring of the given string w in $O(m)$ time.

- Construct the suffix tree T of string w. And match string P along from the root to leaf

- If there exists a complete match, then P is a substring of w, otherwise, not.



Check if "cacac" is a substring of "caaaaccacac"

Applications of Suffix tree

- Besides, there are many other applications of suffix tree.

Given a suffix tree of a string w of length n, …

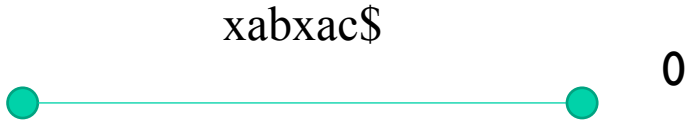
1. Find the first occurrence of the patterns $P_1,\ldots,P_q$, of total length m in $O(m)$ time.
2. Search for a regular expression in P in time expected sublinear in n.
3. Find the longest common substrings of string w_i and w_j in $\Theta(n_i+n_j)$ time.
4. Find the longest repeated substring in $\Theta(n)$ time.
5. …

How to grow a suffix tree (naïve method)

- The running time for a naïve construction of suffix tree is $O(n^2)$

(n: text size)

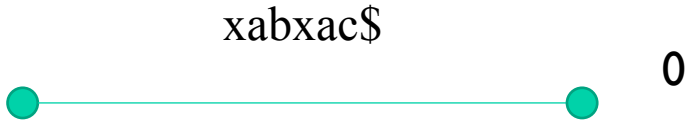
- For example, we want to construct a suffix tree of string "xabxac"



1. Start with the whole string (leaf number 1) and connect the root with the leaf

How to grow a suffix tree (naïve method)

2. Generate suffixes $w[1 \ldots n\text{-}1]\$$, $w[2 \ldots n\text{-}1]\$$, … , $w[n\text{-}1]\$$, and push them into the tree one by one.



Suffixes:

- abxac$
- bxac$
- xac$
- ac$
- c$
- $

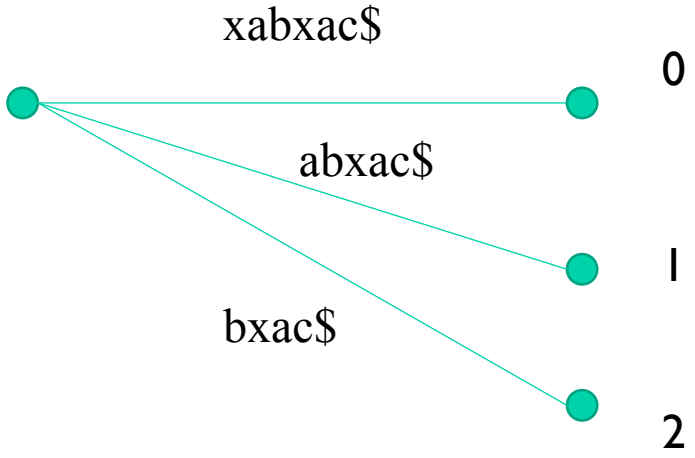
How to grow a suffix tree (naïve method)

3. To insert $Sfx_i = w[i \ldots n\text{-}1]\$$, follow the path from the root, matching characters of Sfx_i until the first mismatch at the character $Sfx_i[j]$.

There are two cases:

i. If the matching cannot continue from a node (which means mismatch happens to be at the beginning of next edge), then create a new node. Label the edge to its corresponding substring.

Insert second and third suffixes



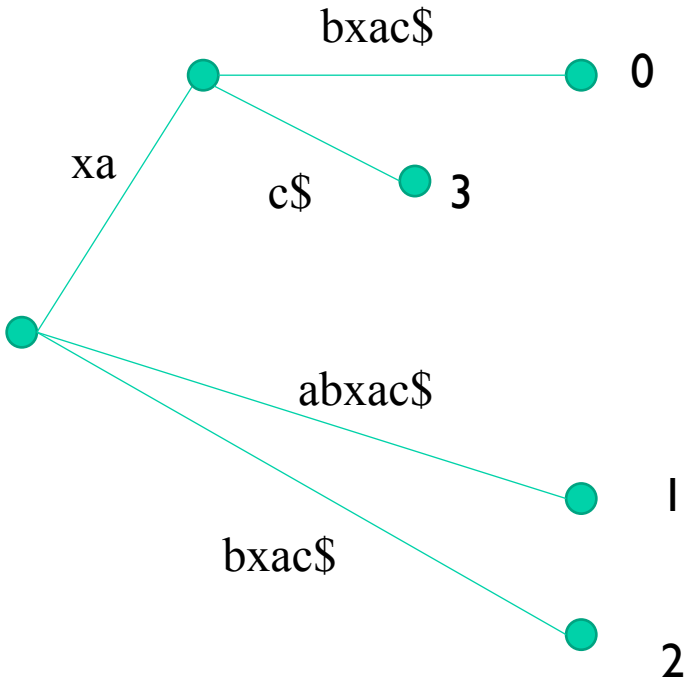
How to grow a suffix tree (naïve method)

ii. If the mismatch occurs in the middle of an edge $e = (u,v)$, then denote the edge to be $a_0, \ldots a_{l-1}$.

Let the mismatch occur at a_k, then create a new node w, and replace edge e by edges (u,w) and (w,v), labeled by $a_1, \ldots, a_{k-1}$, and $a_k \ldots a_{l-1}$.

Then create another new node to store the rest of the newly inserted suffix.

Insertion of "xac$" causes first edge to split

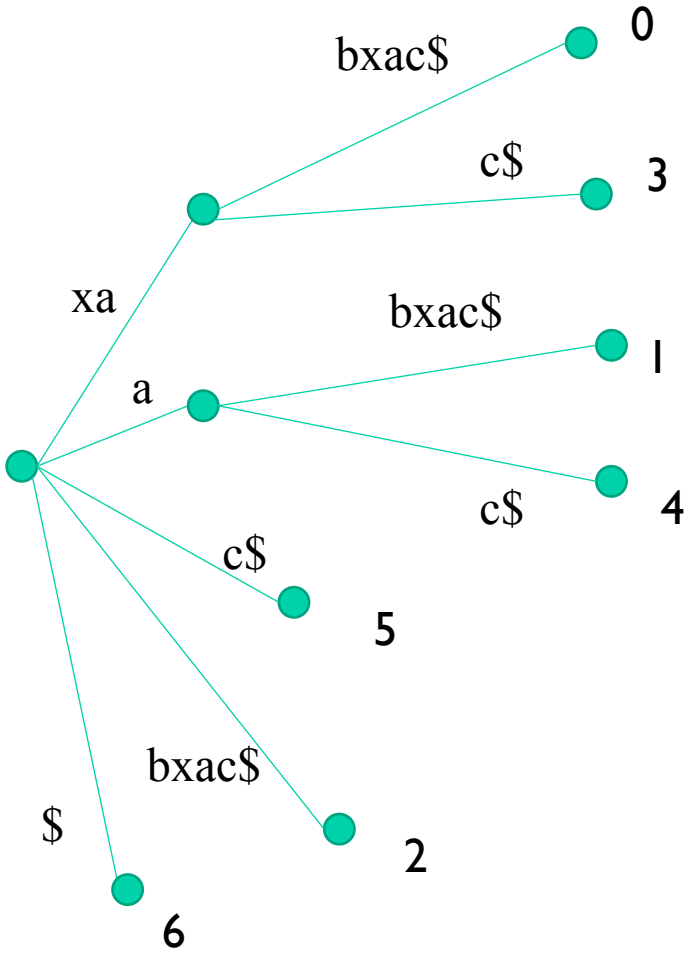


How to grow a suffix tree (naïve method)

Same thing happens when inserting "ac$"

After inserting "ac$" ,"c$" and "$", the suffix tree is complete

Finally, in both cases, a new leaf is created ,numbered *i*.



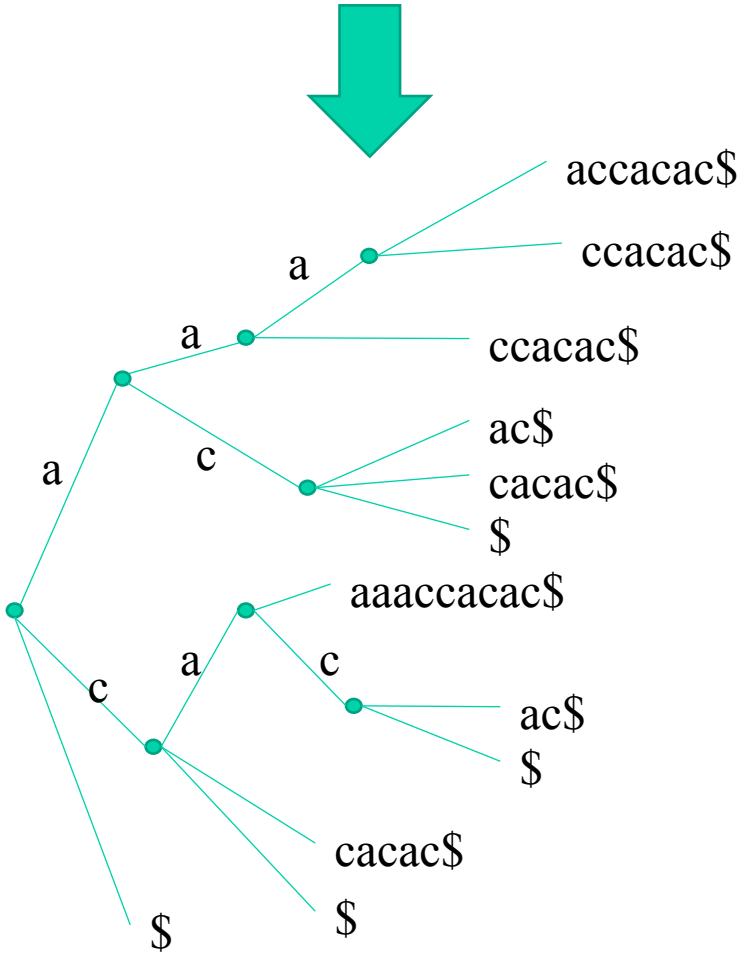
PWM/PSSM using suffix tree

- How can a suffix tree accelerate the process of matching?

(1) We first find the proper length of a target sequence segment. The length can be decided based on memory size.

(2) Then we construct suffix trees from the target sequence.

Suppose we have "caaaaccacac$"



PWM/PSSM using suffix tree

(3) Then a depth-first traversal of the tree is performed, calculated all the prefix scores ($pfxS_d$) for edge labels.

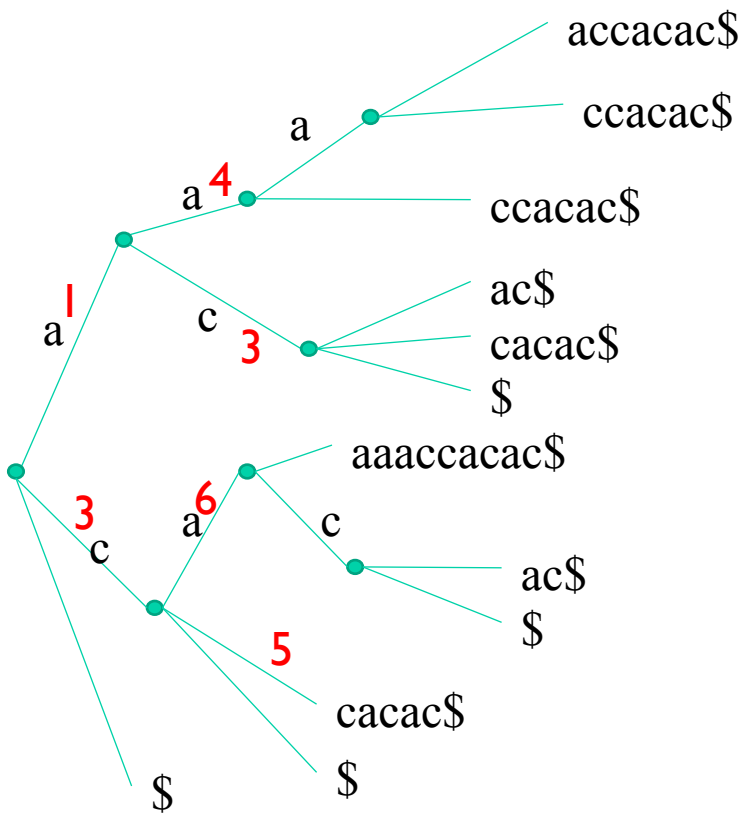
Suppose we have the score functions like the following:

$S_{a,0} = 1 \quad S_{a,1} = 3 \quad S_{c,0} = 3 \quad S_{c,1} = 2$

for a given threshold: $th = 6$

We have intermediate thresholds: th_0=3, th_1=6

Afterwards, we calculate all the prefix scores for edge labels.



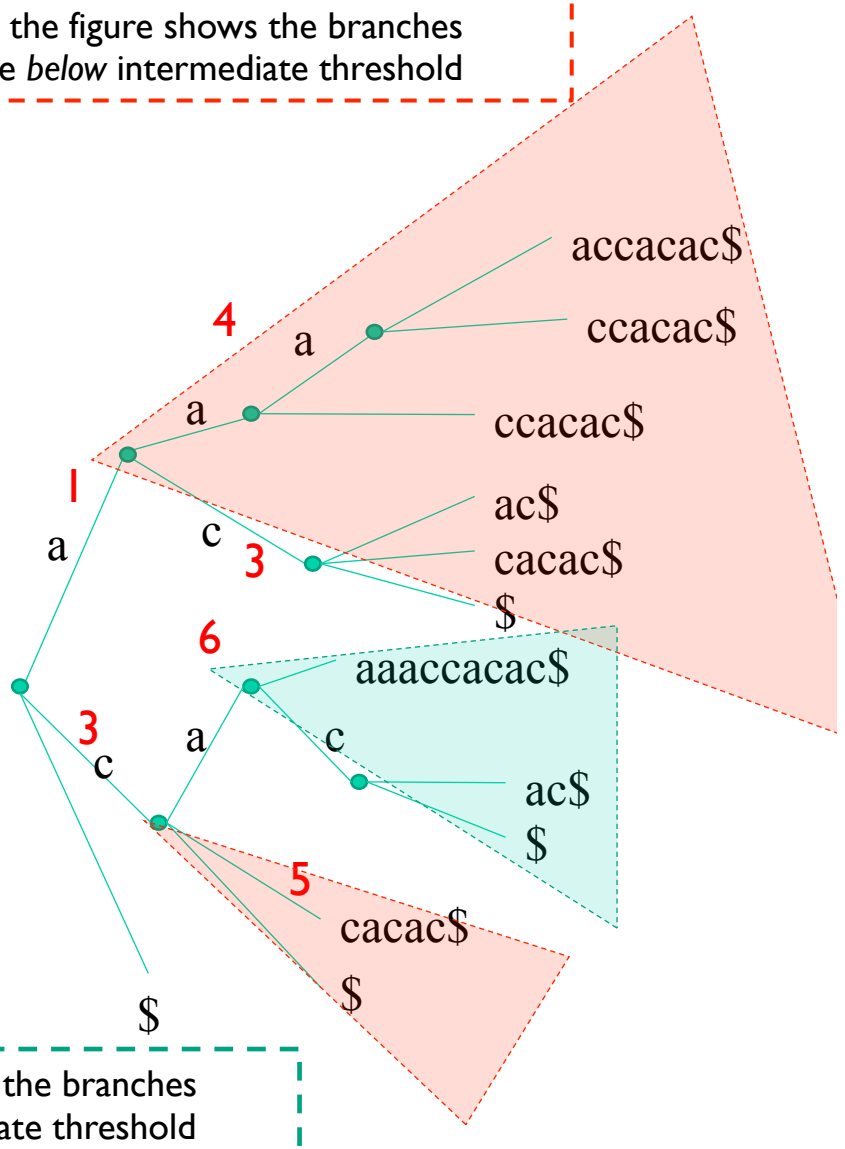
PWM/PSSM using suffix tree

Red zone in the figure shows the branches having score *below* intermediate threshold

(4) Finally analyze the scores, check if either of the two cases happens:

i. Any score at some node in the tree reaches the threshold, then all of its substrings represented by tree reaches the threshold as well.

ii. Similarly, check if any of the scores fall below the *intermediate threshold* , then the whole substring branch can be ignored.



Blue zone in the figure shows the branches having scores *above* intermediate threshold

Suffix tree → Suffix array

Enhanced suffix array

- M. Beckstette *et al.* (2006) brought forward a PWM-based searching method using "enhanced suffix arrays".

- In their study, they focused on the improvement of space efficiency when searching with PWM. Their method is similar to the suffix tree discussed in the previous slides.

- Main features:

Three arrays are kept for different usages:

1. *suf* array

suf array specifies the first indices of each suffix.

2. *lcp* array

lcp array stores the length of the longest common prefix of two adjacent suffixes according to leaf numbers.

3. *skp* array

Sorry, a little bit complex, talk about it in the following slides.

Enhanced suffix array (array *suf*)

suf array specifies the first indices of each suffix..

$S_{suf}[0], S_{suf}[1], \ldots, S_{suf}[n\text{-}1]$ is the sequence of suffixes of S in first index position ascending order, where $S_{suf}[i]=S_{suf[i]} = [i \ldots n\text{-}1]$.

i → index if ordered lexicographically

i	*suf*[*i*]	$S_{suf}[i]$
6	0	caaaaccacac$
0	1	aaaaccacac$
1	2	aaaccacac$
2	3	aaccacac$
4	4	accacac$
9	5	ccacac$
7	6	cacac$
3	7	acac$
8	8	cac$
5	9	ac$
10	10	c$
11	11	$

Enhanced suffix array (array *lcp*)

Array *lcp* is an array range from 0 to n with the following features.

(1) $lcp[0] = 0$

(2) $lcp[i]$ stores the length of the longest common prefix of $S_{suf}[i\text{-}1]$ and $S_{suf}[i]$.

The common prefix of "aaaaccacac" and "aaaccacac" is "aaa", so $lcp[1] = 3$

The common prefix of "aaccacac" and "acac" is "a", so $lcp[3] = 1$

i	$lcp[i]$	$S_{suf}[i]$
0	0	aaaaccacac$
1	3	aaaccacac$
2	2	aaccacac$
3	1	acac$
4	2	accacac$
5	2	ac$
6	0	caaaaccacac$
7	2	cacac$
8	3	cac$
9	1	ccacac$
10	1	c$
11	0	$

Enhanced suffix array (array *skp*)

Array *skp* is in range 0 to n such that

$$skp[i] = \min(\{n+1\} \cup \{j \in [i+1, n] \mid lcp[i] > lcp[j]\})$$

Geometrically, $skp[i]$ denotes the next leaf that does not occur in a substree below the branching node corresponding to the longest common prefix of $S_{suf}[i\text{-}1]$ and $S_{suf}[i]$.

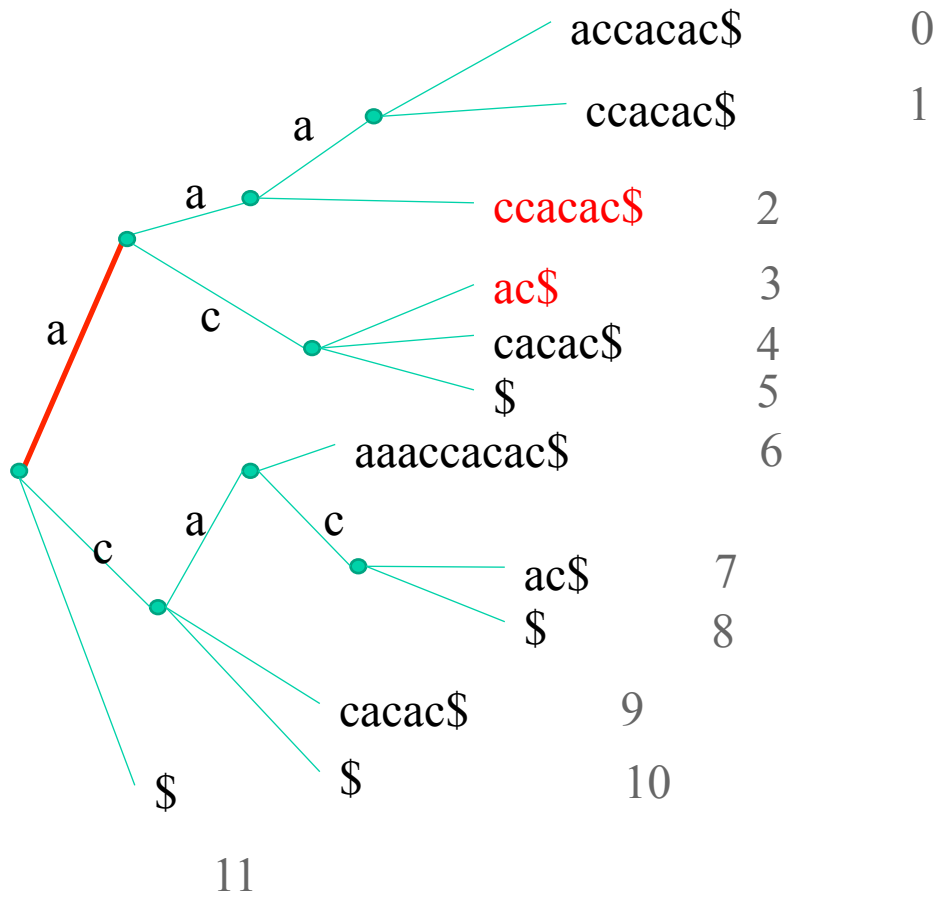
→ *skp[i]* is the next index j where where *lcp[j] < lcp[i]*

i	*lcp[i]*	*skp[i]*	$S_{suf}[i]$
0	0	12	aaaaccacac$
1	3	2	aaaccacac$
2	2	3	aaccacac$
3	1	6	acac$
4	2	6	accacac$
5	2	6	ac$
6	0	12	caaaaccacac$
7	2	9	cacac$
8	3	9	cac$
9	1	11	ccacac$
10	1	11	c$
11	0	12	$

Enhanced suffix array (array *skp*)

Longest common prefix of "acac$" and "aaccacac$" is "a", so *lcp*[3] = 1.

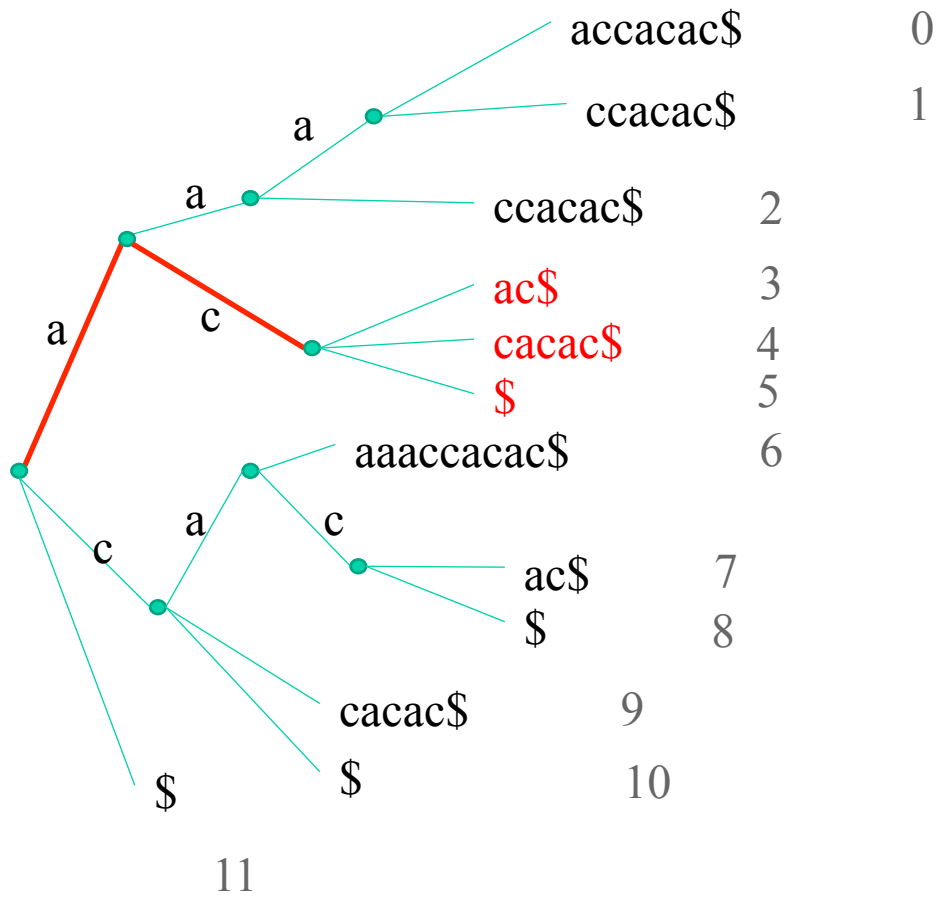
The red edge indicates the common prefix.



Enhanced suffix array (array *skp*)

Similarly, we can find out that $lcp[4] = lcp[5] = 2$

The red edge indicates the common prefix.

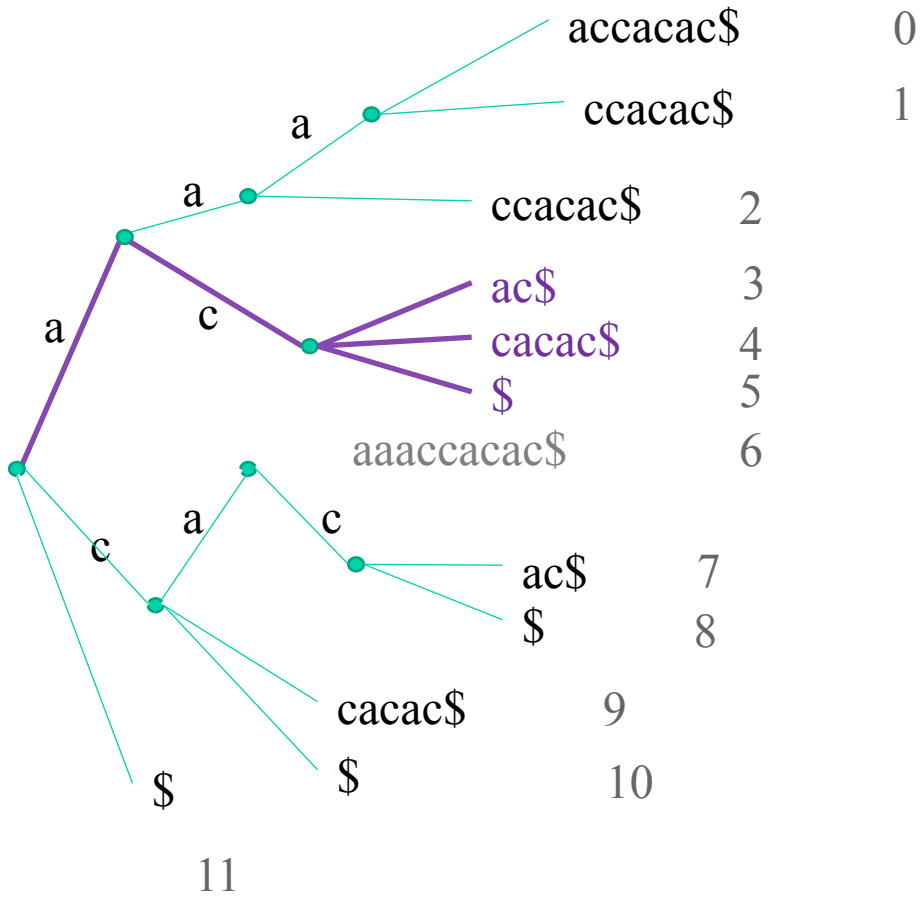


Enhanced suffix array (array *skp*)

We cannot find common prefix between "caaaaccacac$" and "ac$", so $lcp[6]= 0$

Therefore, $skp[3] = skp[4] = skp[5]= 6$.

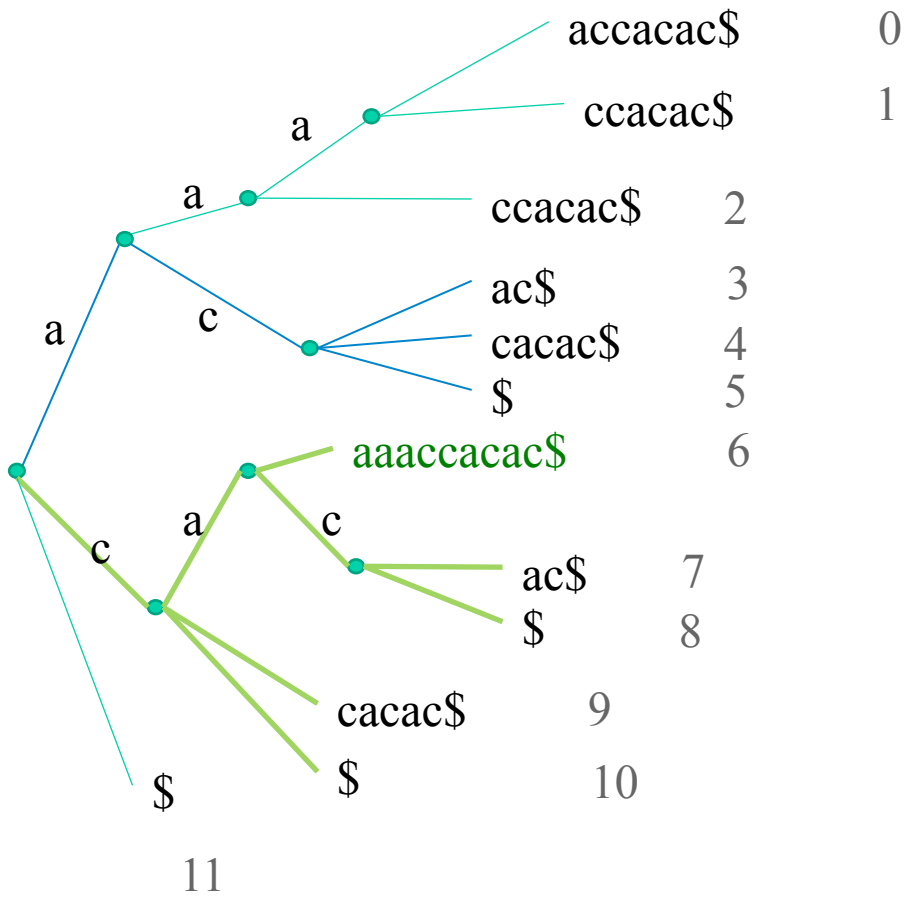
In the graph, we can easily tell $S_{suf}[6]$ is the *first node* (colored in green) not occurring in branch of $S_{suf}[3]$, $S_{suf}[4]$ and $S_{suf}[5]$ (colored in purple).



Enhanced suffix array (array *skp*)

Starting from "caaaaccacac$" no node occurs in another branch (branch not involved with the current suffix).

Therefore, *skp*[6]=12



References

- A.E. Kel *et al*. MATCHTM: a tool for searching transcription factor binding sites in DNA sequences. (2003) *Nucleic Acids Research Vol. 31 No. 13*
- M. Beckstette *et al.* PoSSuMsearch: Fast and Sensitive Matching of Position Specific Scoring Matrices using Enhances Suffix Arrays (2004)
- M. Beckstette *et al.* Fast Index based algorithms and software for matching position specific scoring matrices. (2006) *BMC Bioinformatics*
- S. Rahmann *et al.* On the Power of Profiles for Transcription Factor Biding Site Detection. (2003) *Statistical Applications in Genetics and Molecular Biology*
- B. Dorohonceanu *et al.* Accelerating Protein Classification Using Suffix Trees. (2000)

Thank you!

Suffix arrays/trees and PWM matching

Enhanced suffix array

- **Definition (1):** prefix score for sequence w

$$pfxS_d(w) = \sum_{j=0}^{d} \ln(f_{w(j),j} / \pi_{w(j)}) \qquad w(j) \in \{A,T,G,C\} \text{ for all } j$$

where w is a sequence segment, $w(j)$ is the character of w at index j.

Denote $l_i = \min\{M\ , |S_{suf}[i]|\ \}\text{-}1.$

- **Definition (2):** d_i as the largest depth of the suffix that satisfies the intermediate threshold

$$d_i = \max(\{-1\} \cup \{d \in [0, l_i] \mid pfxS_d(S_{suf}[i]) \geq th_d\})$$

- **Definition (3):** $C_i[d]$ is the prefix score of $S_{suf}[i]$ with depth d

$$C_i[d] = pfxS_d(S_{suf}[i]) \qquad \text{for all } d \in [0, d_i]$$

Enhanced suffix array

Notice that, for each $S_{suf}[i]$, the following statements are equivalent:

$$d_i = M - 1 \Leftrightarrow$$
$$pfxS_{M-1}(S_{suf}[i]) = C_i[M-1] \geq th_{M-1}$$

M is the length of PWM

That is, $S_{suf}[i]$ satisfies the threshold iff the *largest depth satisfying the intermediate threshold* equals to the **length of the PWM**

We will show the algorithm by an example.
Suppose we have following score functions :

$S_{i,j}$	Index 0	Index 1	Index 2
a	1	3	2
c	3	2	1

Suppose we have following threshold:
$th = 7$
Intermediate thresholds:
$th_0 = 2$, $th_1 = 5$, $th_2 = 7$.

Enhanced suffix array

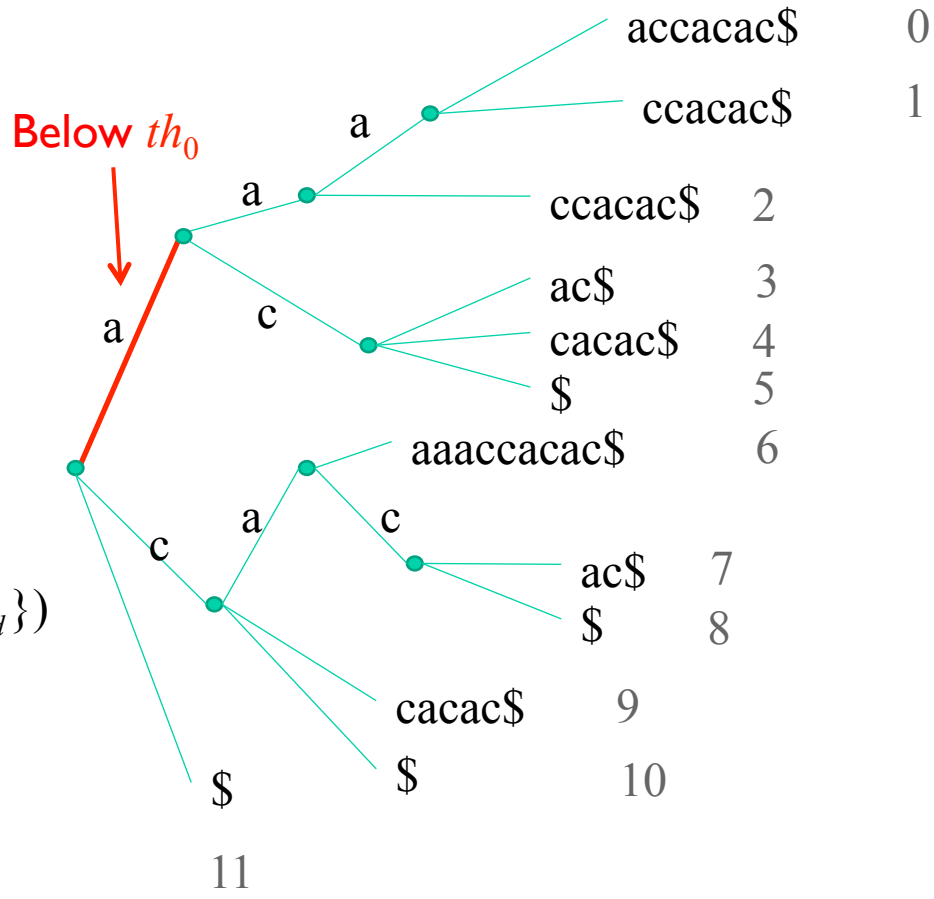
Algorithm:

1. First compute C_0 and d_0 to see if the first suffix satisfies the threshold

For the $S_{suf}[0]$ = "aaaaccacac$", we have $C_0[0] = pfxS_0(S_{suf}[0]) = 1$, below the threshold.

$$d_0 = \max(\{-1\} \cup \{d \in [0, l_0] \mid pfxS_d(S_{suf}[0]) \geq th_d\})$$

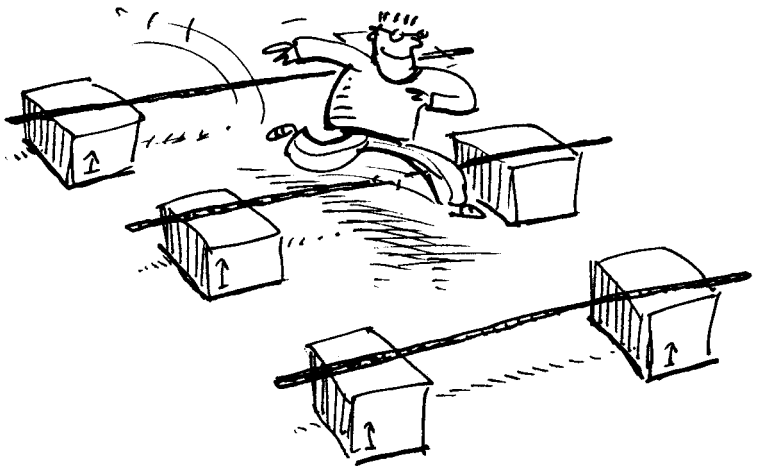
Hence we have d_0 = -1, meaning no prefix satisfies threshold.



Enhanced suffix array

2. Afterwards, it's the VERY tricky part.

Based on the *skp* array, we can acutally JUMP over some suffixes.



By following the rules below:

For each $S_{suf}[i]$ satisfying/not satisfying the threshold,
we try to find the first k that $d_i+1 >= lcp[k]$, by the following jumping cascade:

let $k_0 = i+1$, $k_1 = skp[k_0]$ …, $k_m = skp[k_{m-1}]$ such that,
$d_i + 1 < lcp[k_1]$,
$d_i + 1 < lcp[k_2]$,…,
$d_i + 1 < lcp[k_{m-1}]$
and $d_i+1 >= lcp[k_m]$

k_m is the k we want.
And any suffixes within the jump range satisfy/ do not satisfy the threshold as $S_{suf}[i]$ satisfies/ does not satisfy the threshold

Enhanced suffix array

i. In the first step, we have $d_0 = -1$
ii. We try to find first k such that $d_0+1=0 >= lcp[k]$.
iii. By making three jumps based on skp array, we find $k_3 = 6$ satisfying our case.

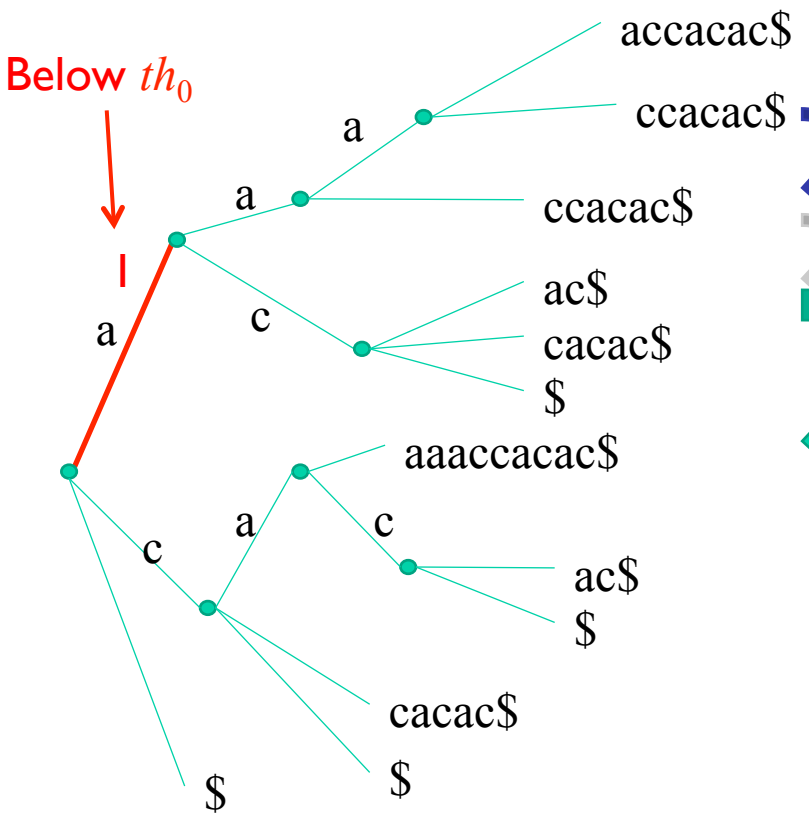
First jump: $k_1 = skp[k_0=0+1=1] = 2$
$d_0+1=0< lcp[k_1] = 2$

Second jump: $k_2 = skp[k_1] = 3$
$d_0+1=0< lcp[k_2] = 1$

Third jump: $k_3 = skp[k_2] = 6$
$d_0+1=0>= lcp[k_3] = 0.$
YEAH, we got it !!!

i	$lcp[i]$	$skp[i]$	$S_{suf}[i]$
0	0	12	aaaaccacac$
1	3	2	aaaccacac$
2	2	3	aaccacac$
3	1	6	acac$
4	2	6	accacac$
5	2	6	ac$
6	0	12	caaaaccacac$
7	2	9	cacac$
8	3	9	cac$
9	1	11	ccacac$
10	1	11	c$
11	0	12	$

Enhanced suffix array



i. In the first step, we have $d_0 = -1$
ii. We try to find first k such that $d_0+1 >= lcp[k]$.
iii. By making three jumps based on skp array, we find $k_3 = 6$ satisfying our case.

First jump: $k_1 = skp[k_0] = 2$
$d_0+1=0< lcp[k_1] = 2$

Second jump: $k_2 = skp[k_1] = 3$
$d_0+1=0< lcp[k_2] = 1$

Third jump: $k_3 = skp[k_2] = 6$
$d_0+1=0>= lcp[k_3] = 0.$
YEAH, we got it !!!

Since $S_{suf}[0]$ does not satisfy the threshold,
$S_{suf}[1]\ldots S_{suf}[5]$ cannot satisfy the threshold.

Enhanced suffix array

Next we compute C_6, d_6,

$$d_i = \max(\{-1\} \cup \{d \in [0, l_i] \mid pfxS_d(S_{suf}[i]) \geq th_d\})$$

$$C_i[d] = pfxS_d(S_{suf}[i]) \qquad \text{for all } d \in [0, d_i]$$

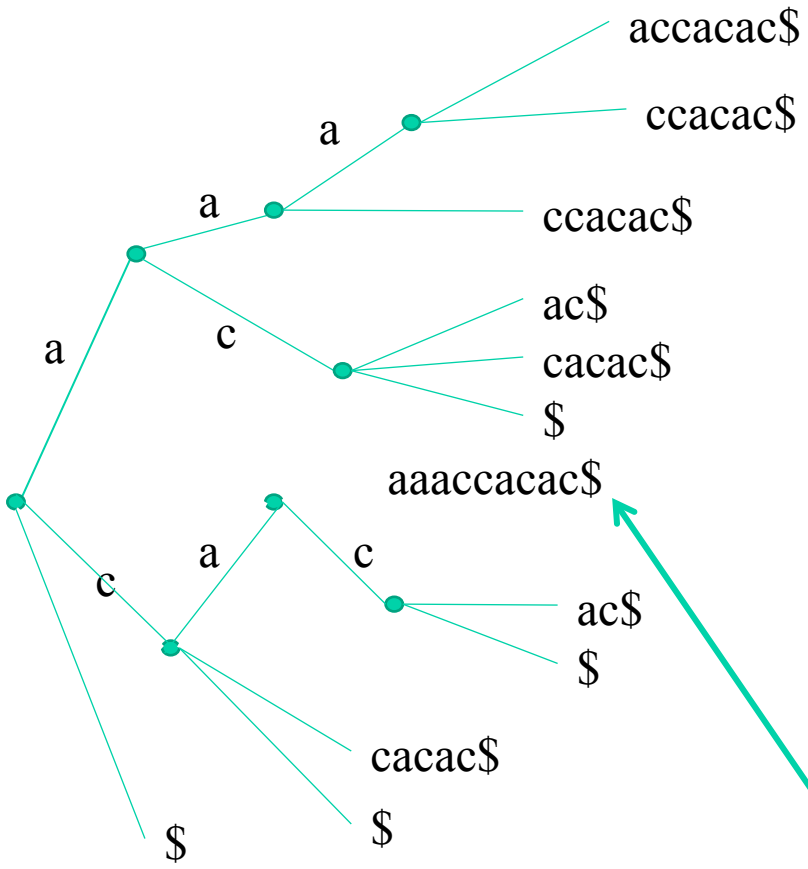
We obtain: $d_6 = 2$ and $C_6[0] = 3$, $C_6[1] = 6$, $C_6[2] = 8$, satisfying all intermediate thresholds. Therefore, $S_{suf}[6]$ is a signal.

$S_{i,j}$	Index 0	Index 1	Index 2
a	1	3	2
c	3	2	1

$S_{suf}[6]$= "caaaaaccacac$"

Suppose we have following threshold:
$th = 7$
Intermediate thresholds:
$th_0 = 2$, $th_1 = 5$, $th_2 = 7$.

Enhanced suffix array



Similarly, we try to find the first k such that $d_6+1=3 >= lcp[k]$.

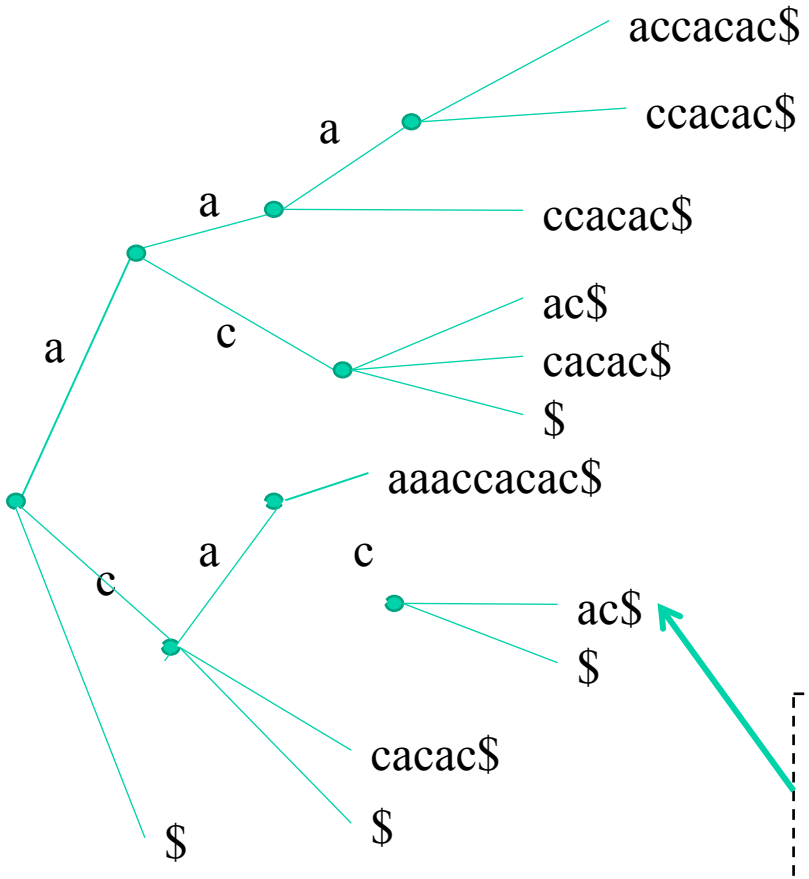
We find that, $k_0=6+1=7$

Satisfying
$d_6+1=2+1=3 >= lcp[k_0=7] = 2$

Therefore, only $S_{suf}[6]$ satisfies the threshold in this round. Next we continue to compute C_7 and d_7

No JUMP here.
Only moves to the next node

Enhanced suffix array



By similar approach, we obtain
$d_7 = 2$ and $C_7[0] = 3$, $C_7[1] = 6$, $C_7[2] = 7$,
satisfying all intermediate threshold.

Similarly, we try to find the first k such that
$d_7+1=3 >= lcp[k]$.

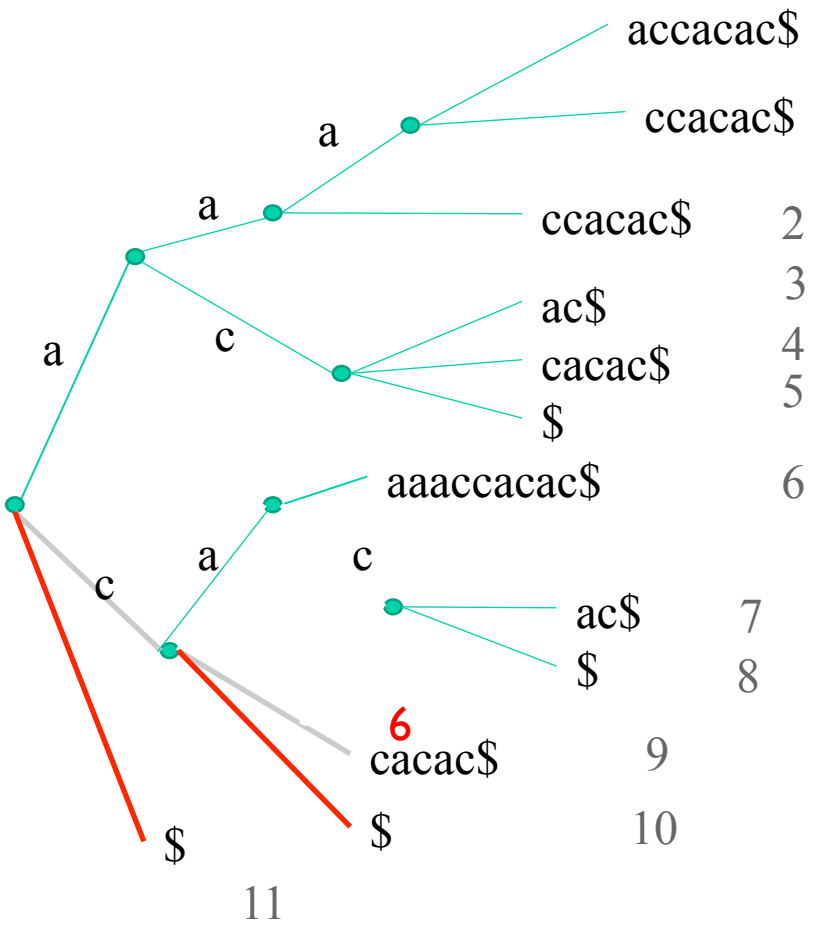
We find that, $k_0=7+1=8$

Satisfiying
$d_7+1=2+1=3 >= lcp[k_0=8] = 3$

Therefore, only $S_{suf}[7]$ satisfies the threshold in this round.

No JUMP here.
Only moves to the next node

Enhanced suffix array



By similar approach, we obtain $S_{suf}[8]$,$S_{suf}[9]$ satisfying the threshold; $S_{suf}[10]$ and $S_{suf}[11]$ not satisfying the threshold

(Algorithm ends)

Enhanced suffix array (algorithm)

1. Compute d_0 , and $C_0[d]$ for any $d \in [0, d_0]$

2. Assume d_{i-1} and $C_{i-1}[d]$ has been determined, then we calculate d_i and $C_i[d]$ from d_{i-1} and $C_{i-1}[d]$:

   Since $S_{suf}[i\text{-}1]$ and $S_{suf}[i]$ have a common prefix of length $lcp[i]$,
   we have, $C_{i-1}[d] = C_i[d]$ for all $d \in [0, lcp[i]-1]$

   To calculate $C_i[d]$ for all $d \in [0, d_i]$, the following two cases need to considered:

   (1) $d_{i-1} + 1 >= lcp[i]$
   Then compute $C_i[d]$ for $d_{i+1} > lcp[i]$ while $d <= l_i$ and $C_i[d] >= th_d$

Enhanced suffix array (algorithm)

(2) $d_{i-1} + 1 < lcp[i]$

Suppose we have j be the minimum value from $[i+1, n+1]$ such that all suffixes $S_{suf}[i], S_{suf}[i+1] \ldots S_{suf}[j-1]$ have a common prefix of length $d_{i-1}+1$.

Then, according to the definition,

i. if $d_{i-1} = m-1$, then there are signals at all position $S_{suf}[r]$ for $i<=r<=j-1$

ii. If $d_{i-1} < m-1$, then no signals for all position $S_{suf}[r]$

We obtain j by following a chain of entries in array skp, computing a chain of values :

$j_0=i, j_1 = skp[j_0] \ldots, j_k = skp[j_{k-1}]$ such that,

$d_{i-1} + 1 < lcp[j_{k-1}]$ and $d_{i-1}+1 >= lcp[j_k]$