Space efficient data structures for pattern matching

Last lecture: Suffix trees O(n) words.

Over 20+ years: O(n) words -> n words -> ... -> O(n) bits

Suffix arrays, ..., compressed suffix arrays, FM-index

- A rare case in TCS literature: Breakthrough theoretical results whose implementations have practical impact.
- Simple (but non-intuitive) data structures, yet surprisingly powerful for pattern matching.
- Space complexity: suffix trees O(n) words, suffix arrays n words, CSA & FM-index O(n) bits

Suffix arrays & compressed suffix arrays

Today's focus: SA & CSA

Tricky / non-trivial, but not complicated:

- Binary search is more powerful that you thought.
- Counter-intuitive: searching backward is better than searching forward
- A sequence of n integers with max value n can be stored using less than $n \log n$ bits if it is an increasing sequence.

Suffix Arrays –Manber & Myers 1993

- Space: n words
- Let T[1..n] be a string of n characters. The suffix array of T is an array, denoted SA[1..n], of n integers.
- SA[i] = j means that
  - the suffix T[j..n] is (lexicographically) the i-th smallest suffix, or equivalently,
  - the rank of T[j..n] is i.

How big is a word? Θ(log n) bits.

Example

T = acaaccg$

For illustration only.
We don't store the suffixes explicitly.

i (rank)	SA[i] (start position)	T[SA[i]..n]
1	8	$
2	3	aaccg$
3	1	acaaccg$
4	4	accg$
5	2	caaccg$
6	5	ccg$
7	6	cg$
8	7	g$

Increasing lexico-order of suffixes

Searching a suffix array

Given a pattern P, we can find its occurrences in T using a binary search on SA.

T = acaaccg$

	i	SA[i]	T[SA[i]..n]	
L →	1	8	$	smallest
	2	3	aaccg$	
	3	1	acaaccg$	
M = (L+R)/2 →	4	4	accg$	
	5	2	caaccg$	
	6	5	ccg$	
	7	6	cg$	
R →	8	7	g$	largest

Example

Consider P = “aa”. M = 4, and T[4..5] > P.

T = acaaccg$

SA[M] = 4

	i	SA[i]	T[SA[i]..n]
L →	1	8	$
	2	3	aaccg$
	3	1	acaaccg$
M = (L+R)/2 →	4	4	accg$
	5	2	caaccg$
	6	5	ccg$
	7	6	cg$
R →	8	7	g$

Example

Consider P = "aa". T[3..4] = P.

T = acaaccg$

SA[M] = 3

	i	SA[i]	T[SA[i]..n]
L →	1	8	$
M →	2	3	aaccg$
	3	1	acaaccg$
R →	4	4	accg$
	5	2	caaccg$
	6	5	ccg$
	7	6	cg$
	8	7	g$

Binary search

- To locate the first occurrence of P in SA:
  - if P ≤ T[SA[M]..n] then R = M; else L = M
- Last occurrence:
  - if P < T[SA[M]..n] then R = M; else L = M

P = "ac"

First occc → row 3

Last occ → row 4

i	SA[i]	T[SA[i]..n]
1	8	$
2	3	aaccg$
3	1	acaaccg$
4	4	accg$
5	2	caaccg$
6	5	ccg$
7	6	cg$
8	7	g$

Time Complexity

A binary search involves comparing P with log n suffixes, each comparison examines at most m = |P| characters.

To locate the region of the SA table containing P requires O(m log n) time.

It takes O(1) time to report the position of each occurrence.

P = ac

i	SA[i]	T[SA[i]..n]
1	8	$
2	3	aaccg$
3	1	acaaccg$
4	**4**	accg$
5	2	caaccg$
6	5	ccg$
7	6	cg$
8	7	g$

Can we do better?

O(m + log n) time is feasible if we keep an addition of 2n LCP values. (I.e., SA + LCP require 3n words.)

Consider any integers i and j. LCP(i,j) is the length of the longest prefix of the suffixes starting at SA[i] and SA[j].

E.g.,

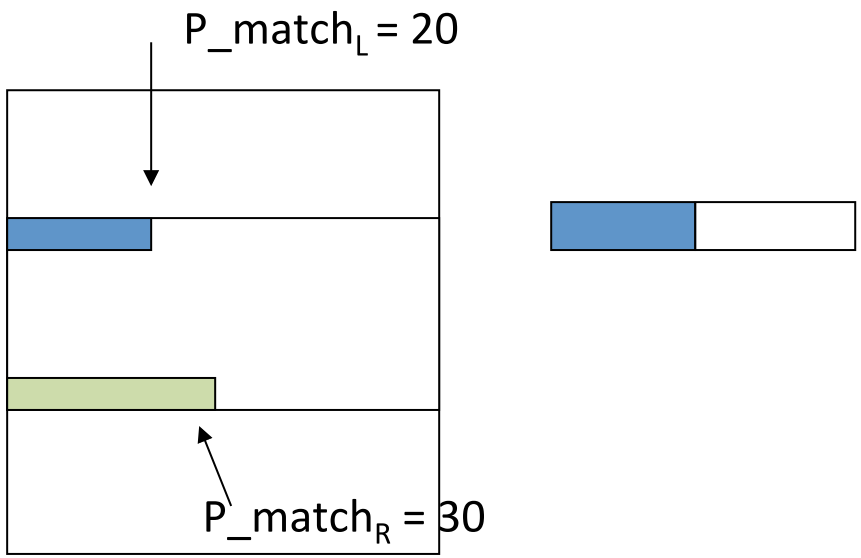
i	SA[i]	T[SA[i]..n]
1	8	$
2	3	aaccg$
3	1	acaaccg$
4	4	accg$
5	2	caaccg$
6	5	ccg$
7	6	cg$
8	7	g$

LCP(3, 4) = 2

LCP(5, 8) = 0

How many possible LCP values? n(n-1)/2

Initial setting



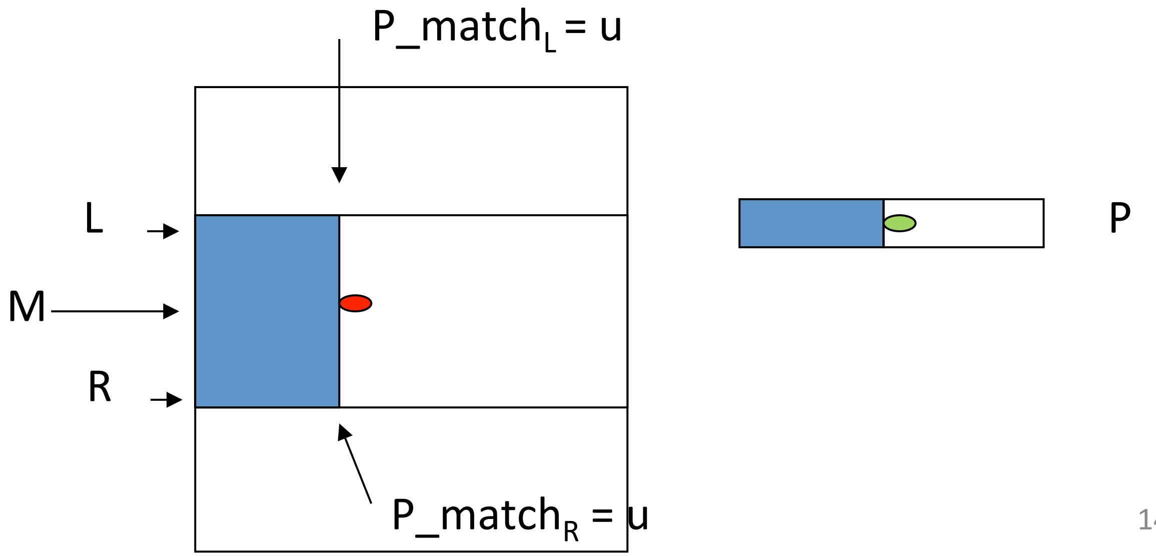
- L = 1; R = n
- Store additional information:
  - the number of characters P matches the boundary suffixes T[SA[L]..n] and T[SA[R]..n]).
- Initially, when L = 1, R = n,
  - P_match_L = length of longest common prefix (P, "$");
  - P_match_R = length of longest common prefix (P, T[SA[n]..n]).
  - it takes O(m) extra time to compute P_match_L and P_match_R
- In each subsequent iteration, it takes O(1) time to update P_match_L and P_match_R.

Pattern matching: simple case

Background: In each iteration, we compute M = (L+R)/2 and compare P against the suffix T[SA[M]..n].

Assume the simple case: $P\_match_L = P\_match_R = u$.

- The first u characters of the suffixes at SA[L], SA[L+1], SA[L+2], …, SA[R] are all the same as P.
- When we compare the suffix T[SA[M]..n] with P, we start the comparison at character P[u + 1] instead of P[1].



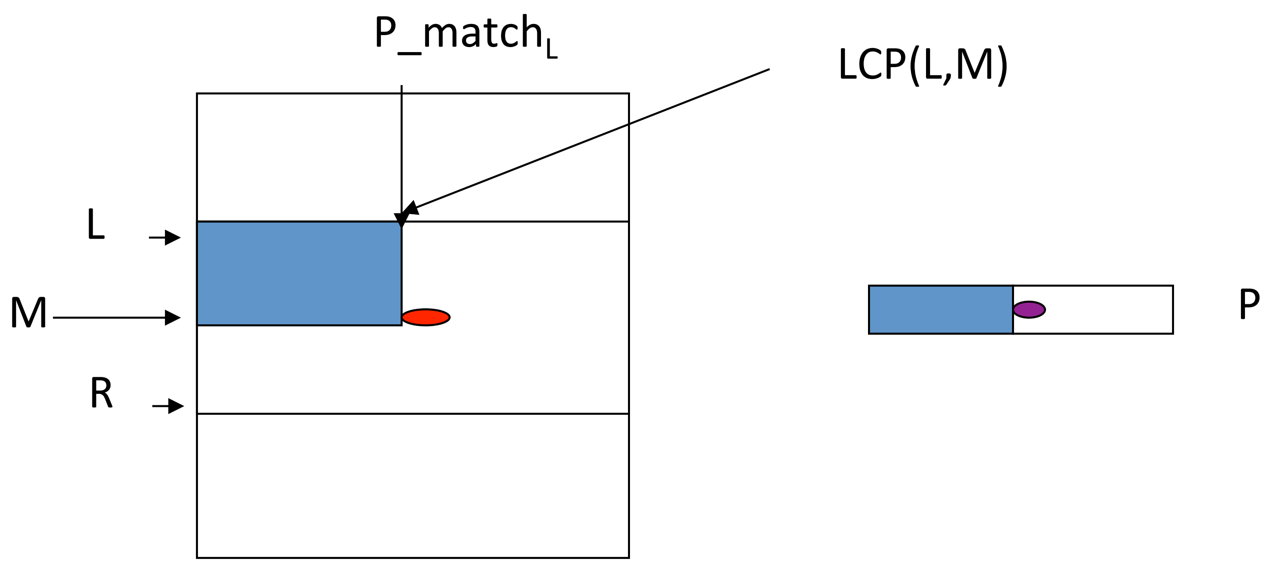
General case: $P\_match_L \neq P\_match_R$

- Assume that $P\_match_L > P\_match_R$
  - **Exercise:** $P\_match_L < P\_match_R$
- Compute LCP(L, M)   [Assume O(1) time.]
  - LCP(L, M) : longest common prefix of the suffixes at SA[L] and SA[M];
  - P_match_L : longest common prefix of P and the suffix at SA[L].
- THREE cases:
  1. LCP(L, M) = P_match_L
  2. LCP(L, M) < P_match_L
  3. LCP(L, M) > P_match_L

$P\_match_L > P\_match_R$ and Case 1

$LCP(L, M) = P\_match_L$

- P and T[SA[M]..n] have P_match_L characters in common.
- Compare P and the suffix T[SA[M]..n] starting from position $P\_match_L + 1$.
- Update L or R (P_match_L or P_match_R, resp.) depending on whether T[SA[M]..n] < P or not.

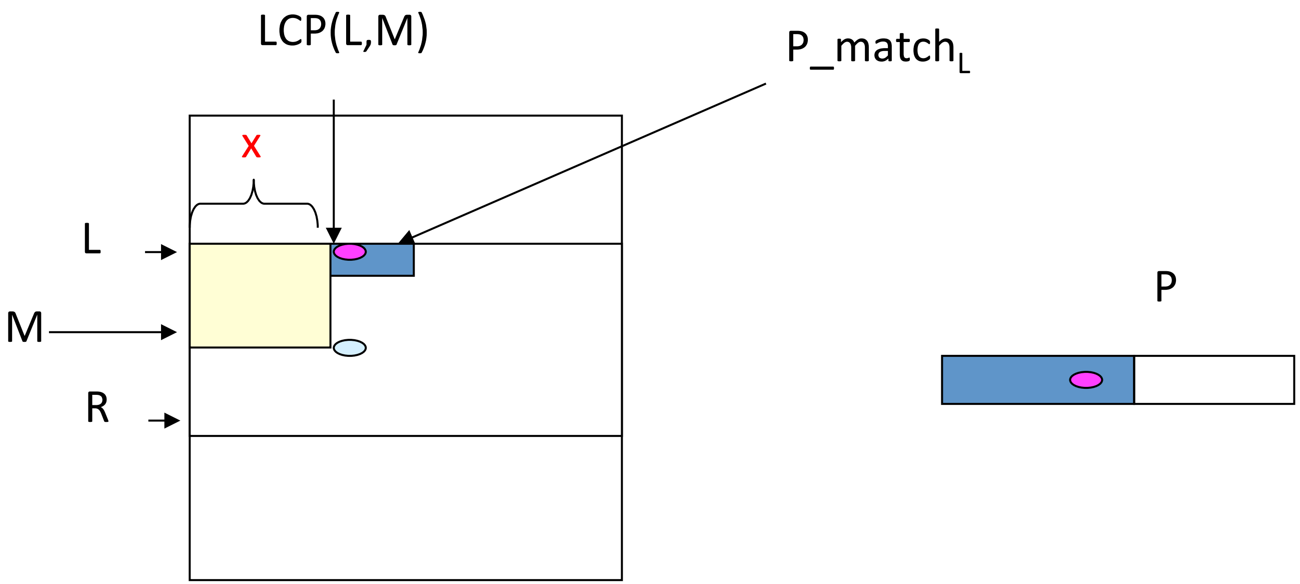


$P\_match_L > P\_match_R$ and Case 2

LCP(L, M) < P_match_L

Zero character comparison !

- Let x = LCP(L,M).
  - Then T[SA[M] + x] > T[SA[L] + x] = P[1+x].
  - I.e., the suffix starting at SA[M] is larger than P.
- Without any further comparison, we can immediately update R = M, and P_match_R = x.

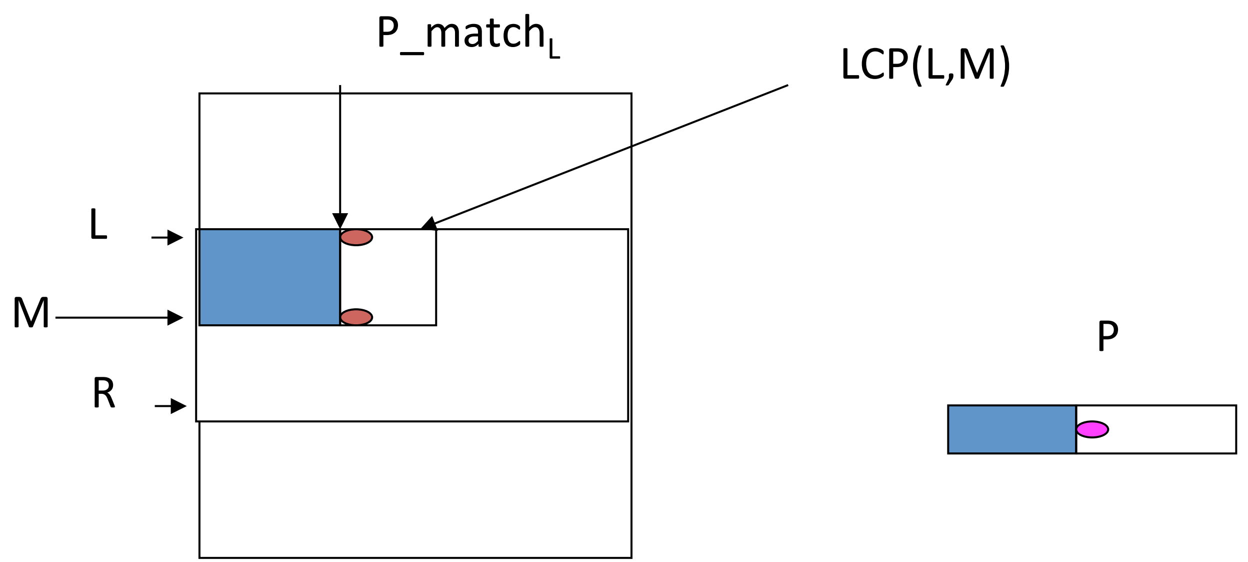


$P\_match_L > P\_match_R$ and Case 3

Zero character comparison !

$LCP(L, M) > P\_match_L$:

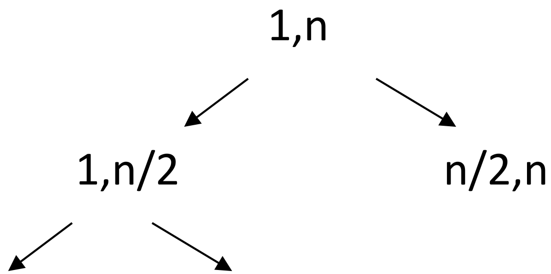
Set $L = M$; P_match_L remains the same



O(m + log n) time & 2n LCP values

- The binary search still requires log n phases. Yet the character comparison of P never goes backward.

- We don't need all possible LCP values. (Otherwise, there are n^2 LCP values). The LCP values required can be defined by a binary tree modeling the binary search. There are at most 2n LCP values.



Compressed Suffix arrays (CSA)

- $O(n)$ bits (instead of $O(n)$ words), assuming the alphabet size Σ is a constant.
- Grossi & Vitter, STOC 2000; Sadakane, ISAAC 2000.

- Recall that $SA[i]$ is the starting position of the i-th smallest suffix (with rank = i).

  $SA^{-1}[j]$ is the rank of the suffix starting at position j.
- Let ψ be any array defined as follows:

  $\Psi[i] = SA^{-1}[SA[i] + 1]$.
- Intuitively, we look at the i-th smallest suffix and delete its first character, $\Psi[i]$ is the rank of the resulting suffix.

Example

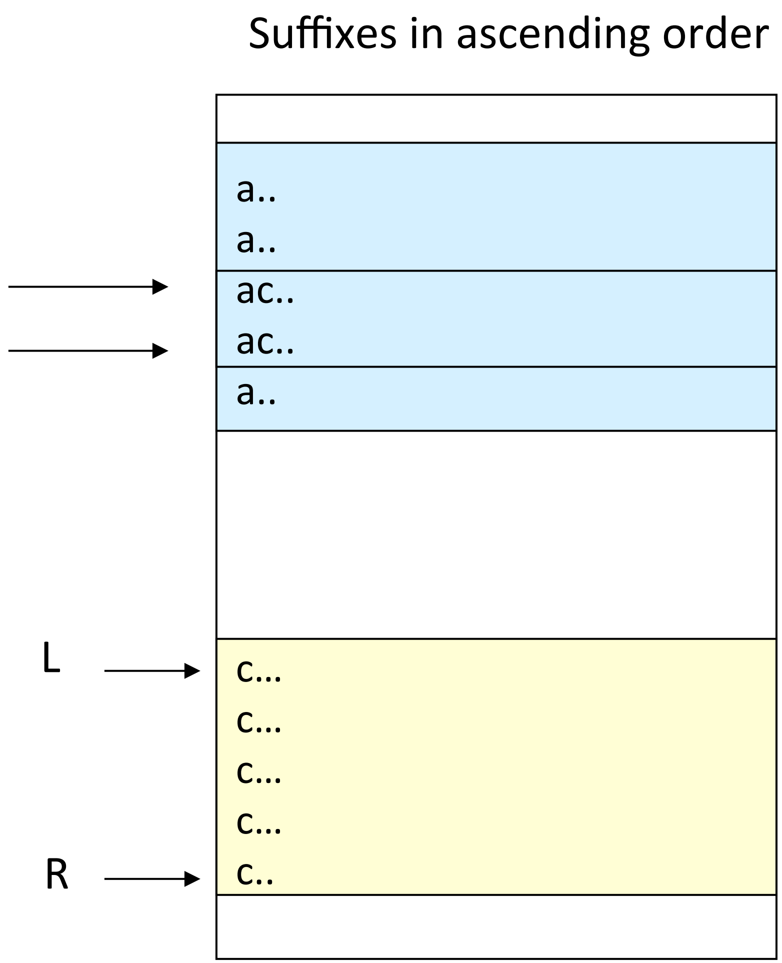
$\Psi[i] = SA^{-1}[SA[i] + 1]$

i	SA[i]	Ψ[i]	T[SA[i]..n]
1	8	3	$
2	3	4	aaccg$
3	1	5	acaaccg$
4	4	6	accg$
5	2	2	caaccg$
6	5	7	ccg$
7	6	8	cg$
8	7	1	g$

Boundary case: $\Psi[1] = SA^{-1}[T[1..n]]$.

Backward searching with CSA

Pattern matching: O(m log n + occ log n) time, where m = |P|.



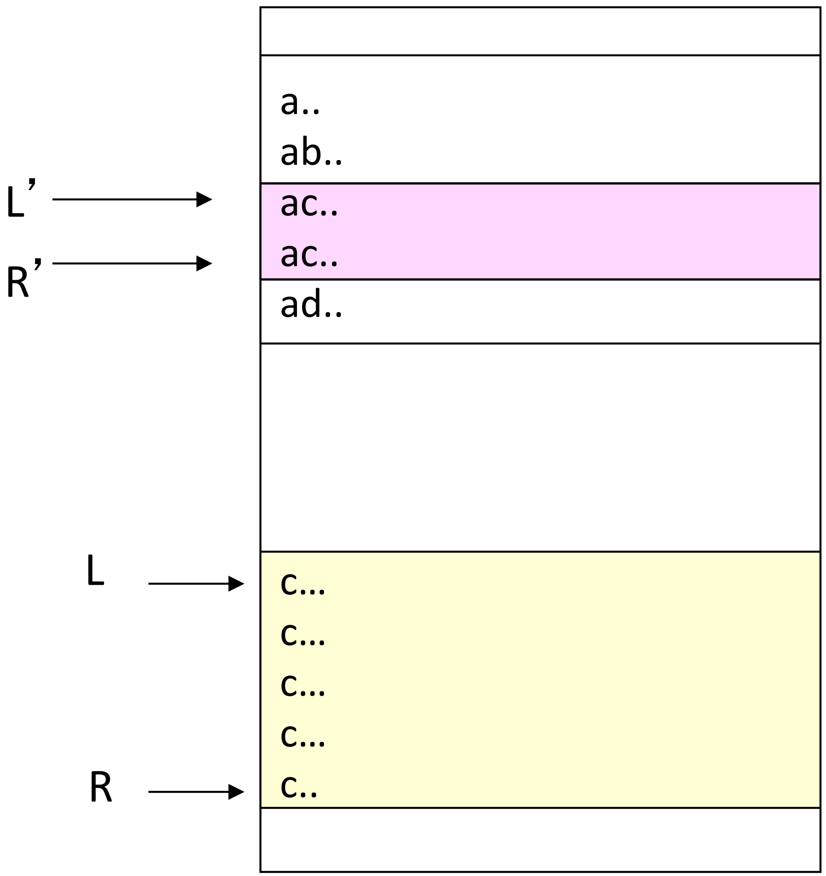
Consider P = “ac”.

Starting with the last character of P, i.e., “c”,
suppose we’ve found the region of suffixes starting with “c”.

How to find the region starting with “ac”?

Backward searching with CSA

Suffixes in ascending order



Observation about L’ and R’.

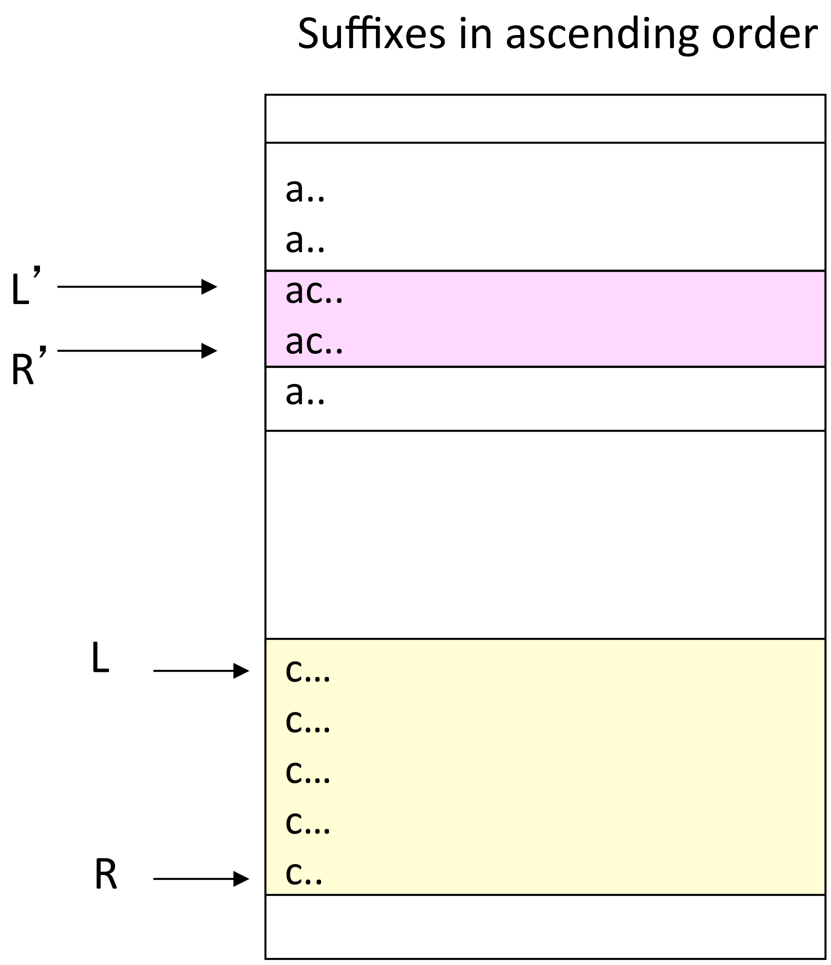
$\Psi[L' - 1] < L$

$L \le \Psi[L'] \le R$

$L \le \Psi[R'] \le R$

$\Psi[R' + 1] > R$

Binary search again



Within the region starting with "a",

L' is the smallest index such that Ψ[L'] is inside the range [L,R], and R' is the biggest index with Ψ[R'] inside [L,R].

I.e., L' & R' can be found using binary search w.r.t. Ψ.
It takes O(log n) time.

Details

Pre-compute, for each character c in the alphabet,

count(c) = # of characters < c in the string T being indexed.

- count is a small array; space = $O(\Sigma \log n)$ bits.
- The region [count(c)+1, count(c')] in the SA table stores all suffixes with 1st character = c, where c' is the next character lexicographically after c.

Details

Pe-compute, for each character c in the alphabet, count(c) = # of characters < c in the string T being indexed.

- count is a small array; space = $O(\Sigma \log n)$ bits.
- The region [count(c)+1, count(c')] in the SA table stores all suffixes with 1st character = c, where c' is the next character lexicographically after c.

Let P[1..m] be a given pattern.

Let c = P[m], and c' = the 1st character lexicographically after c.

L = count (c) + 1; R = count(c').

For i = m-1 to 1

- Let c = P[i], and let c' = the next character after c.
- Within the range [count(c)+1, count(c')], use binary search to find
  - the smallest index L' with Ψ[L'] in [L,R], and
  - the biggest index R' with Ψ[R'] in [L,R].
- L = L', and R = R'

Space Complexity

- Consider all suffixes starting with the same character, say, "a".
- Suppose their ranks are in the range i, i+1, i+2, …., j.
- Lemma. $\Psi[i] < \Psi[i+1] < \Psi[i+2] < \ldots < \Psi[j]$.
- Proof.
  - Consider the i-th and the (i+1)-th smallest suffix.
  - They have the same first character. Thus, excluding the first character, the i-th smallest suffix is still smaller than the (i+1)-th smallest suffix.
  - Thus, $\Psi[i] < \Psi[i+1]$.

Representing an increasing sequence

Suppose that $1 \leq \Psi[i] < \Psi[i+1] < \Psi[i+2] < \ldots < \Psi[j] \leq n$.

- Trivial representation of $\Psi[i] \mathinner{..} \Psi[j]$
  - $n' \log n$ bits, where $n' = j-i+1$.

- Sampling: store only ONE out of every log n values;
  - space : $(n'/\log n) \log n$ bits $= n'$ bits.
  - How to retrieve those values which are NOT sampled.

Representing an increasing sequence

Suppose that $1 \le \Psi[i] < \Psi[i+1] < \Psi[i+2] < \ldots < \Psi[j] \le n$.

- And store the difference:
  - $\Psi[i+1] - \Psi[i]$
  - $\Psi[i+2] - \Psi[i+1]$
  - $\Psi[i+3] - \Psi[i+2]$
  - …
  - $\Psi[j] - \Psi[j-1]$
- As Ψ is monotonic increasing, the sum of these differences is $\Psi[j] - \Psi[i] \le n$.
- We store each difference d as a unary number: a 1 followed by d 0's.
  - E.g., 4, 2, 3... → 100001001000 …
  - The entire bit sequence contains n' 1's and at most n 0's

Difference bit vector

Given a bit vector representing $d_1, d_2, d_3, \ldots$

100001001000….

We can build an index such that

- for any k, we can compute in O(1) time
  - the sum of $d_1, d_2, \ldots, d_k$,
  - or equivalently, the # of zeros before the (k+1)-th one

Rank & Select

- Raman et al. SODA 2002
- Given a bit vector *A* of length b, we can represent *A* using o(b) bits* to support the rank and the select operation in O(1) time:
  - Select (i): find the position of the i-th 1's.
  - Rank(h): find the number of 1's before A[h].
- Build an rank-select index over the difference bit vector 10010000010000001100010…,
  - we can compute in O(1) time the sum of $d_1, d_2, \ldots, d_k$ using the formula: Select(k+1) – k.
  - We can also compute $d_i, d_{i+1}, \ldots, d_j$ in O(1) time.

* (1 + o(1)) b log log b/log b + O(b/log b) bits

Remarks

- Rank & Select are complicated data structures.
  - a good theoretical result, but real implementation is slower than expected.
- Next lecture: practical solution
  - table look up
  - hardware: SSE instructions

References

- Survey paper: Compressed full-text indices, Gonzalo Navarro and Veli Mäkinen , ACM Computing surveys, 2007. (http://portal.acm.org/citation.cfm?id=1216370.1216372)