Suffix array

Array of suffixes in lexicographic order

(assume $\$ < a \quad \forall a \in \Sigma$)

i	0	1	2	3	4	5	6
$S[i]$	b	a	n	a	n	a	$

i	$SA[i]$	Suffix
0	**6**	$
1	**5**	a$
2	**3**	ana$
3	**1**	anana$
4	**0**	banana$
5	**4**	na$
6	**2**	nana$

i	0	1	2	3	4
$S[i]$	a	a	a	a	$

i	$SA[i]$	Suffix
0	**4**	$
1	**3**	a$
2	**2**	aa$
3	**1**	aaa$
4	**0**	aaaa$

Suffix array

- Array of suffixes in lexicographic order
- Simpler structure, continuous memory
- Less memory: one index per character ($4n$ bytes in total)
- Construction in $O(n)$ even for large alphabets
- Search for a pattern P by binary search in $O(m \log n)$ time, can be improved to $O(m + \log n)$ with additional memory

Suffix trees and arrays - summary

Data structure	Search	# pointers/integers
Suffix tree	$O(m \log \sigma)$	$7n$ or more
Suffix array	$O(m \log n)$	n
Suffix array + LCP	$O(m + \log n)$	$3n$

Today:

- Construction of suffix arrays
- Computation of lcp values
- Next week: construction of suffix trees from suffix arrays

How to create a suffix array

Goal: sort all suffixes lexicographically

Not so good options:

- Use MergeSort. Running time?
- Use RadixSort. Running time?
- Create a suffix tree, then convert to array. How?

Instead use $O(n)$ algorithm, e.g. Karkkainen and Sanders 2003

Inverse of a suffix array

Array rank such that $rank[i] = x \iff SA[x] = i$

Can be computed in $O(n)$ from SA:

```
1  for(i=0;i<=n,i++){
2     rank[SA[i]] = i;
3  }
```

Go from suffix to its position in the suffix array, its neighbors, etc.

Master theorem

One of three cases:

If $T(n) = aT(\frac{n}{b}) + f(n)$
where $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$
and $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ for some $c < 1$,
then $T(n) = \Theta(f(n))$.

In our algorithm: $T(n) = T(\frac{2}{3}n) + O(n)$
$a = 1, b = \frac{3}{2}$
$\log_{\frac{3}{2}}(1) = 0$
$a \cdot f(\frac{n}{b}) = \frac{2}{3}n$
Therefore $T(n) = \Theta(n)$.

Longest common prefix

- $\mathrm{lcp}(A, B) =$ the length of longest common prefix of strings A and B
- $\mathrm{LCP}(i, j) = \mathrm{lcp}(T[SA[i]..n-1], T[SA[j]..n-1])$
- $L[i] = \mathrm{LCP}(i, i+1)$

  i.e. lcp of two consecutive suffixes in a suffix array

Lemma: If $SA[x+1]+1 = SA[y+1]$, then $L[y] \geq L[x]-1$.

Suffix	i	$SA[i]$	$L[i]$
$	0	5	0
aabab$	1	0	1
ab$	2	3	2
abab$	3	1	0
b$	4	4	1
bab$	5	2	-

Computation of the LCP array

```
1   h = 0;
2   for(i=0; i<=n; i++) {
3     if(rank[i]>0) {
4       k = SA[rank[i]—1];
5       // compare suffixes S[i..n—1] a S[k..n—1]
6       // assuming they have at least h characters in common
7       while (T[i+h]==T[k+h]) { h++; }
8       // we have found first mismatch
9       L[rank[i]—1] = h;
10      if(h>0) { h——; }
11    }
12  }
```

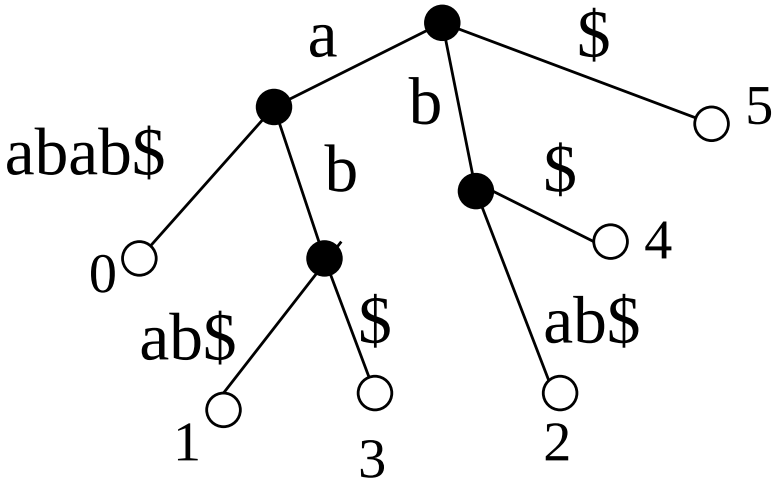
Similar but quadratic-time algorithm

```
1   for(i=0; i<=n; i++){
2     if(rank[i]>0){
3       k = SA[rank[i]—1];
4       // compare suffixes starting at i a k
5       h = 0;
6       while (T[i+h]==T[k+h]) { h++; }
7       // we have found the first difference
8       L[rank[i]—1] = h;
9     }
10  }
```

From suffix array to suffix tree

T =aabab$

Suffix	i	SA[i]	L[i]
$	0	5	0
aabab$	1	0	1
ab$	2	3	2
abab$	3	1	0
b$	4	4	1
bab$	5	2	-



From suffix array to suffix tree

```
1   create roof and leaf w corresponding to SA[0]
2   v = w;
3   for(int i=1; i<=n; i++) {
4      while(v.parent.string_depth>L[i−1]) {
5        v = v.parent;
6      }
7      if(v.parent.string_depth<L[i−1]) {
8        split edge from v.parent to v with a new vertex
9        at string depth L[i−1]
10     }
11     attach new leaf w for SA[i] from v.parent;
12     v = w;
13  }
```