Sorting Suffixes

Juha Kärkkäinen, Peter Sanders

MPI für Informatik, U. Karlsruhe, U. Helsinki

Some Stringology-Speak

String S**:** Array $S[0:n) := S[0{:}n-1] := [S[0],\ldots,S[n-1]]$ of characters

Suffix: $S_i := S[i..n)$

End markers: $S[n] := S[n+1] := \cdots := 0$

0 is smaller than all other characters

Suffix sorting problem

Sort the set $\{S_0, S_1, \ldots, S_{n-1}\}$ of the suffixes of a string S of length n (alphabet $[1,n] = \{1,\ldots,n\}$) into the lexicographic order.

- suffix $S_i = S[i,n]$ for $i \in [0 : n-1]$

$$S = \text{banana}$$

0	banana	$\Longrightarrow$	5	a
1	anana		3	ana
2	nana		1	anana
3	ana		0	banana
4	na		4	na
5	a		2	nana

Suffix sorting problem

Sort the set $\{S_0, S_1, \ldots, S_{n-1}\}$ of the suffixes of a string S of length n (alphabet $[1,n] = \{1,\ldots,n\}$) into the lexicographic order.

- suffix $S_i = S[i,n]$ for $i \in [0 : n-1]$

Applications

- full text indexing
- Burrows-Wheeler transform (`bzip2` compressor)
- replacement for more complex suffix tree

Full text search

Search pattern $P[0:m)$ in text $S[0:n)$ using suffix array SA of S.

Binary search: $O(m \log n)$ good for short patterns

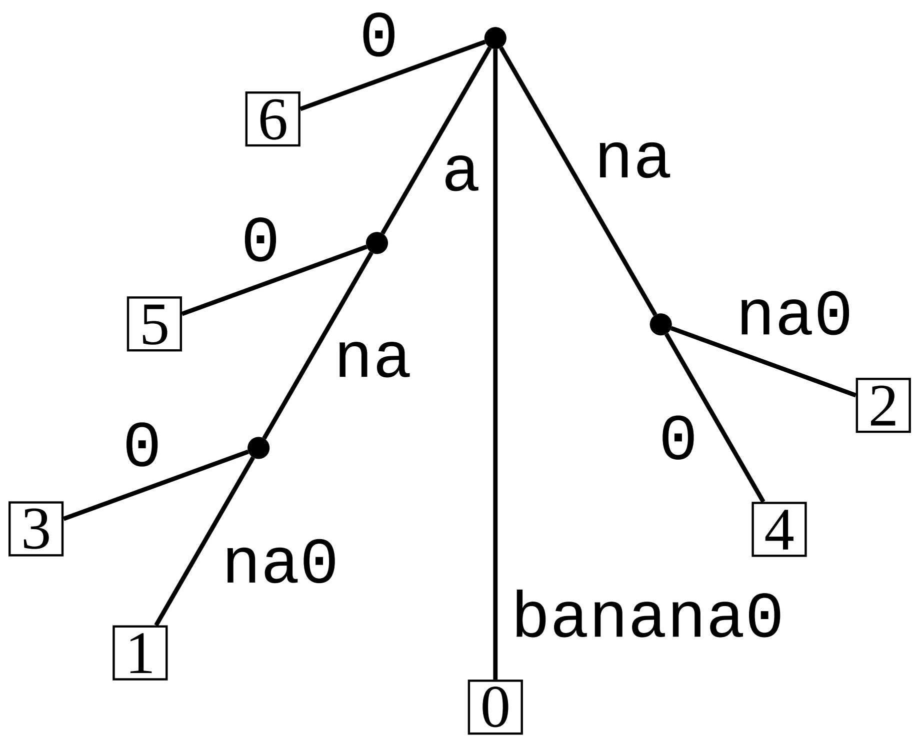
Binary search with lcp: $O(m + \log n)$ if we precompute the longest common prefix between compared strings

Suffix tree: $O(m)$ can be build from SA

Suffix tree [Weiner ’73][McCreight ’76]

- compact trie of the suffixes
- $+$ $O(n)$ time [Farach 97] for integer alphabets
- $+$ Most potent tool of stringology?
- $-$ Space consuming
- $-$ Efficient construction is compli-cated

$S =$ `banana0`



Alphabet model

Ordered alphabet

- only comparisons of characters allowed

Constant alphabet

- ordered alphabet of constant size
- multiset of characters can be sorted in linear time

Integer alphabet

- alphabet is $\{1,\ldots,\sigma\}$ for integer $\sigma \geq 2$
- multiset of k characters can be sorted in $O(k+\sigma)$ time

Ordered $\rightarrow$ *Integer Alphabet*

sort the characters of S

replace $S[i]$ by its rank among the characters

```
012345    125024
banana -> aaabnn
213131 <- 111233
```

Generalization Lexicographic Naming

sort the k-tuples $S[i:i+k)$ for $i \in 1:n$

replace $S[i]$ by the rank of $S[i:i+k)$ among the tuples

A First Divide-and-Conquer Approach

1. $SA^1 =$sort $\{S_i : i$ is odd$\}$ (recursion)
2. $SA^0 =$sort $\{S_i : i$ is even$\}$ (easy using SA^1)
3. merge SA^0 and SA^1 (very difficult)

Problem: its hard to compare odd and even suffixes.
[Farach 97] developed a linear time suffix tree construction algorithm based on that idea. Very complicated.

Was only known linear time algorithm for suffix arrays

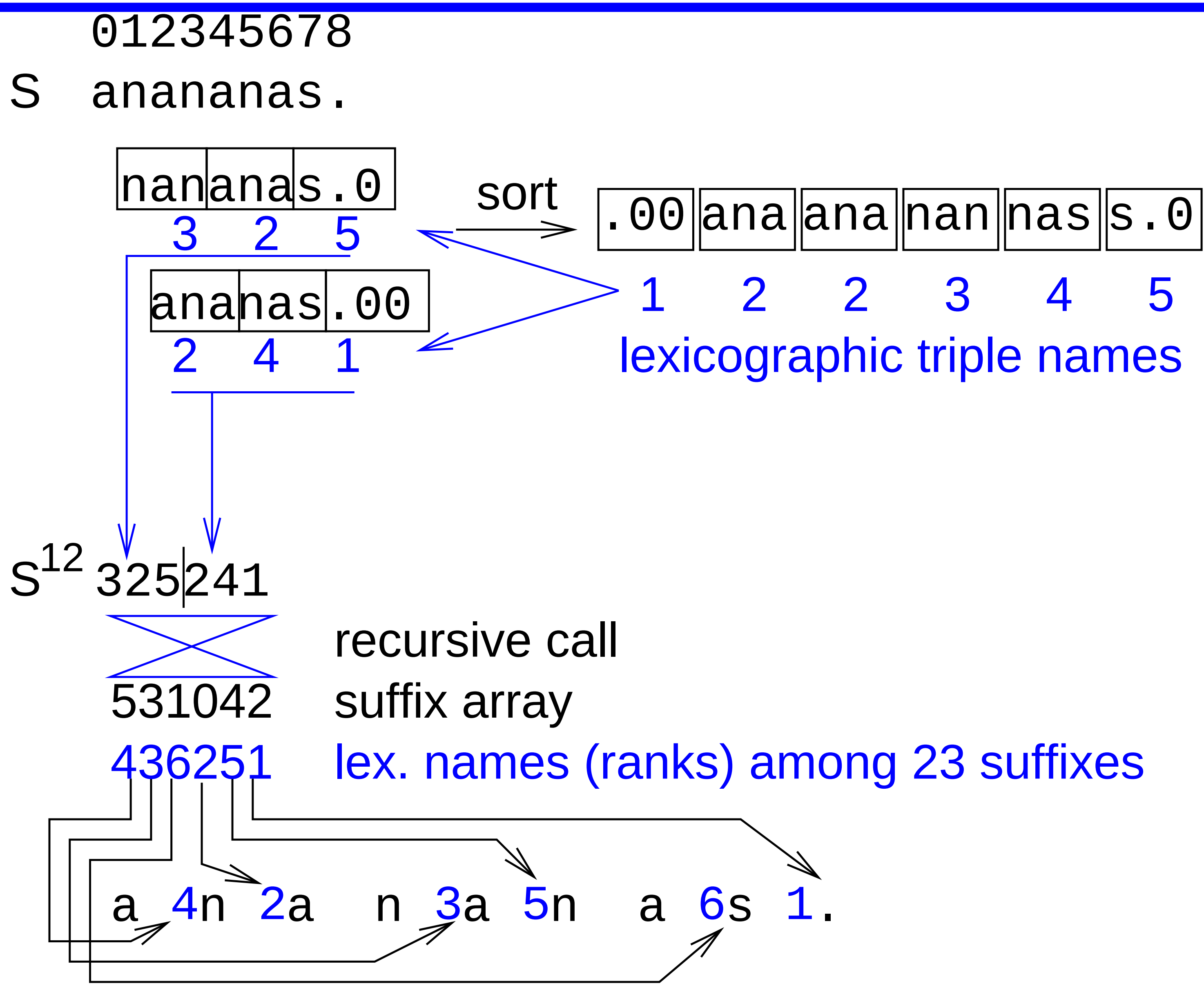
Skewed Divide-and-Conquer

1. $SA^{12} =$sort $\{S_i : i \bmod 3 \neq 0\}$ (recursion)
2. $SA^{0} =$sort $\{S_i : i \bmod 3 = 0\}$ (easy using SA^{12})
3. merge SA^{12} and SA^{0} (easy!)

$$S = \text{banana}$$

$$\begin{array}{|c|l} 5 & \text{a} \\ & \\ 1 & \text{anana} \\ & \\ 4 & \text{na} \\ 2 & \text{nana} \end{array} \quad + \quad \begin{array}{|c|l} & \\ 3 & \text{ana} \\ & \\ 0 & \text{banana} \\ & \\ & \end{array} \quad \Longrightarrow \quad \begin{array}{|c|l} 5 & \text{a} \\ 3 & \text{ana} \\ 1 & \text{anana} \\ 0 & \text{banana} \\ 4 & \text{na} \\ 2 & \text{nana} \end{array}$$

Recursion Example



Recursion

- sort triples $S[i : i+2]$ for $i \bmod 3 \neq 0$
  (LSD-first radix sort)
- find lexicographic names $S'[1 : 2n/3]$ of triples,
  (i.e., $S'[i] < S'[j]$ iff $S[i : i+2] < S[j : j+2]$)
- $S^{12} = [S'[i] : i \bmod 3 = 1] \circ [S'[i] : i \bmod 3 = 2]$,
  suffix S^{12}_i of S^{12} represents S_{3i+1}
  suffix $S^{12}_{n/3+i}$ of S^{12} represents S_{3i+2}
- recurseOn(S^{12}) (alphabet size $\leq 2n/3$)
- Annotate the 23-suffixes with their position in rec. sol.

Least Significant Digit First Radix Sort

Here: Sort n 3-tuples of integers $\in [0:n]$ in lexicographic order

Sort by 3rd position

Elements are sorted by pos 3

Sort stably by 2nd position

Elements are sorted by pos 2,3

Sort stably by 1st position

Elements are sorted by pos 1,2,3

Stable Integer Sorting

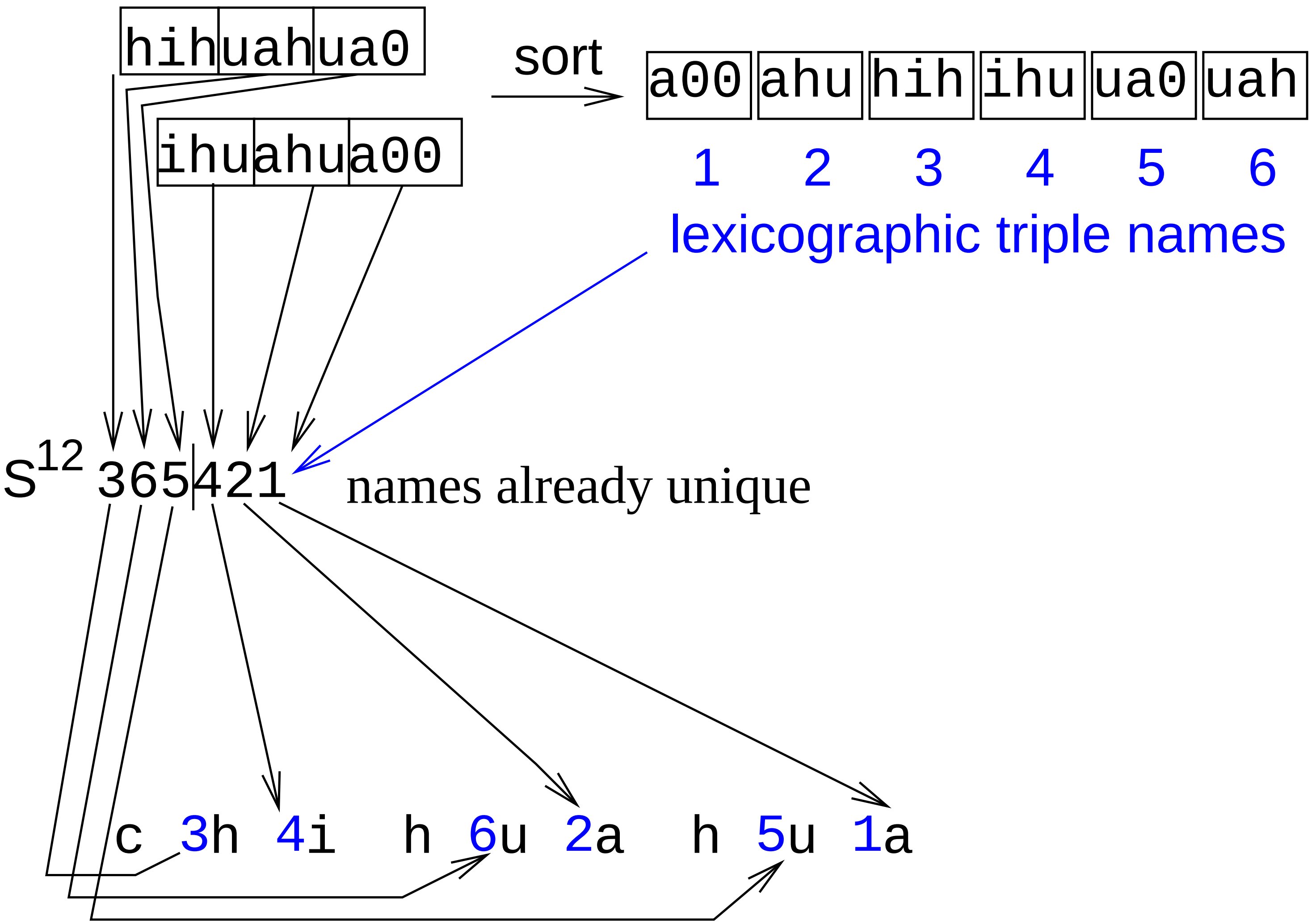
Sort $a[0:n)$ to $b[0:n)$ where $\mathrm{key}(a[i]) \in [0:n]$

$c[0:n] := [0,\ldots,0]$ counters
for $i \in [0:n)$ do $c[a[i]]{+}{+}$ count
$s := 0$
for $i \in [0:n)$ do $(s, c[i]) := (s + c[i], s)$ prefix sums
for $i \in [0:n)$ do $b[c[a[i]]{+}{+}] := a[i]$ bucket sort

Time $O(n)$!

Recursion Example: Easy Case

```
  012345678
S chihuahua
```



Sorting mod 0 Suffixes

0	c $_3$(h $_4$i h $_6$u $_2$a h $_5$u $_1$a)
1	
2	
3	h $_6$(u $_2$a h $_5$u $_1$a)
4	
5	
6	h $_5$(u $_1$a)
7	
8	

Use radix sort (LSD-order already known)

Merge SA^{12} *and* SA^{0}

$0 < 1 \Leftrightarrow \; c\,n < c\,n$
$0 < 2 \Leftrightarrow cc\,n < cc\,n$

3: h 6 u 2 (ahua)	4: (6)u 2 (ahua)
6: h 5 u 1 (a)	7: (5)u 1 (a)
0: c 3 h 4 (ihuahua)	2: (4)i h 6 (uahua)
	1: (3)h 4 (ihuahua)
	5: (2)a h 5 (ua)
	8: (1)a 0 0 0 (0)

$\Downarrow$

```
8:  a
5:  ahua
0:  chihuahua
1:  hihuahua
6:  hua
3:  huahua
2:  ihuahua
7:  ua
4:  uahua
```

Analysis

1. Recursion: $T(2n/3)$ plus

   Extract triples: $O(n)$ (forall $i, i \bmod 3 \neq 0$ do . . .)

   Sort triples: $O(n)$

   (e.g., LSD-first radix sort — 3 passes)

   Lexicographic naming: $O(n)$ (scan)

   Build recursive instance: $O(n)$ (forall names do . . .)

2. $SA^0 =$sort $\{S_i : i \bmod 3 = 0\}$: $O(n)$ (1 radix sort pass)

3. merge SA^{12} and SA^0: $O(n)$ (ordinary merging with strange comparison function)

All in all: $T(n) \leq cn + T(2n/3)$

$\Rightarrow T(n) \leq 3cn = O(n)$

Implementation: Comparison Operators

```
inline bool leq(int a1, int a2,   int b1, int b2) {
  return(a1 < b1 || a1 == b1 && a2 <= b2);
}
inline bool leq(int a1, int a2, int a3,   int b1, int b2, int b3) {
  return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3));
}
```

Implementation: Radix Sorting

```
// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r
static void radixPass(int* a, int* b, int* r, int n, int K)
{ // count occurrences
  int* c = new int[K + 1];                              // counter array
  for (int i = 0;  i <= K;  i++) c[i] = 0;             // reset counters
  for (int i = 0;  i < n;  i++) c[r[a[i]]]++;        // count occurences
  for (int i = 0, sum = 0;  i <= K;  i++) { // exclusive prefix sums
     int t = c[i];  c[i] = sum;  sum += t;
  }
  for (int i = 0;  i < n;  i++) b[c[r[a[i]]]++] = a[i];      // sort
  delete [] c;
}
```

Implementation: Sorting Triples

```
void suffixArray(int* s, int* SA, int n, int K) {
  int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
  int* s12  = new int[n02 + 3];  s12[n02]= s12[n02+1]= s12[n02+2]=0;
  int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
  int* s0   = new int[n0];
  int* SA0  = new int[n0];

  // generate positions of mod 1 and mod  2 suffixes
  // the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1
  for (int i=0, j=0;  i < n+(n0-n1);  i++) if (i%3 != 0) s12[j++] = i;

  // lsb radix sort the mod 1 and mod 2 triples
  radixPass(s12 , SA12, s+2, n02, K);
  radixPass(SA12, s12 , s+1, n02, K);
  radixPass(s12 , SA12, s  , n02, K);
```

Implementation: Lexicographic Naming

```
// find lexicographic names of triples
int name = 0, c0 = -1, c1 = -1, c2 = -1;
for (int i = 0;  i < n02;  i++) {
  if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2) {
    name++;  c0 = s[SA12[i]];  c1 = s[SA12[i]+1];  c2 = s[SA12[i]+2];
  }
  if (SA12[i] % 3 == 1) { s12[SA12[i]/3]      = name; } // left half
  else                  { s12[SA12[i]/3 + n0] = name; } // right half
```

Implementation: Recursion

```
// recurse if names are not yet unique
if (name < n02) {
  suffixArray(s12, SA12, n02, name);
  // store unique names in s12 using the suffix array
  for (int i = 0;  i < n02;  i++) s12[SA12[i]] = i + 1;
} else // generate the suffix array of s12 directly
  for (int i = 0;  i < n02;  i++) SA12[s12[i] - 1] = i;
```

Implementation: Sorting mod 0 Suffixes

```
for (int i=0, j=0;  i < n02;  i++) if (SA12[i] < n0) s0[j++] = 3*SA12[i];
radixPass(s0, SA0, s, n0, K);
```

Implementation: Merging

```
  for (int p=0,  t=n0-n1,  k=0;  k < n;  k++) {
#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
    int i = GetI(); // pos of current offset 12 suffix
    int j = SA0[p]; // pos of current offset 0  suffix
    if (SA12[t] < n0 ?
        leq(s[i],       s12[SA12[t] + n0], s[j],       s12[j/3]) :
        leq(s[i],s[i+1],s12[SA12[t]-n0+1], s[j],s[j+1],s12[j/3+n0]))
    { // suffix from SA12 is smaller
      SA[k] = i;  t++;
      if (t == n02) { // done --- only SA0 suffixes left
        for (k++;  p < n0;  p++, k++) SA[k] = SA0[p];
      }
    } else {
      SA[k] = j;  p++;
      if (p == n0)  { // done --- only SA12 suffixes left
        for (k++;  t < n02;  t++, k++) SA[k] = GetI();
      }
    }
  }
  delete [] s12; delete [] SA12; delete [] SA0; delete [] s0;
}
```

Generalization: Difference Covers

A difference cover D modulo v is a subset of $[0,v)$ such that for all $i \in [0,v)$ there exist $j,k \in D$ with $i \equiv k - j \pmod{v}$.
Example:
$\{1,2\}$ is a difference cover modulo 3.
$\{1,2,4\}$ is a difference cover modulo 7.

- Leads to space efficient variants
- Faster for small alphabets

Improvements / Generalization

- tuning
- use larger difference covers
- external memory implementation
- parallel implementation
- combine with best algorithms for easy inputs
  [Manzini Ferragina 02, Schürmann Stoye 05]

Suffix Array Construction: Conclusion

- simple, direct, linear time suffix array construction
- easy to adapt to advanced models of computation
- generalization to cycle covers yields space efficient implementation

Future/Ongoing Work

- Implementation (internal/external/parallel)
- Large scale applications