A Tale of Three Algorithms:
Linear Time Suffix Array Construction

Juha Kärkkäinen

Department of Computer Science

University of Helsinki

10 August, 2006

5th Estonian Summer School in Computer and Systems Science (ESSCaSS'06)

Linear time suffix array construction

Contents

- Introduction
  - the problem
  - significance
  - history
- Three algorithms from June 2003
  - description in parallel
  - differences and similarities

Suffix array construction

Sort the suffixes of a text lexicographically

- text $T = T[0, n) = t_0 t_1 \cdots t_{n-1}$
- suffix $S_i = T[i, n) = t_i t_{i+1} \cdots t_{n-1}$

Output: suffix array

- sorted array of suffixes
- suffix S_i is represented by i

0 1 2 3 4 5
banana

0	banana
1	anana
2	nana
3	ana
4	na
5	a
6	

$\Longrightarrow$

0 1 2 3 4 5
banana

6	
5	a
3	ana
1	anana
0	banana
4	na
2	nana

Applications

- Full-text indexing
  - binary and backward search
- Construction of other index structures
  - suffix tree
  - compressed indexes
- Text compression
  - Burrows-Wheeler transform
- Finding regularities
  - longest repetition, etc.
- Comparing two or more strings
  - $T = T_1 \# T_2$

0 1 2 3 4 5
banana

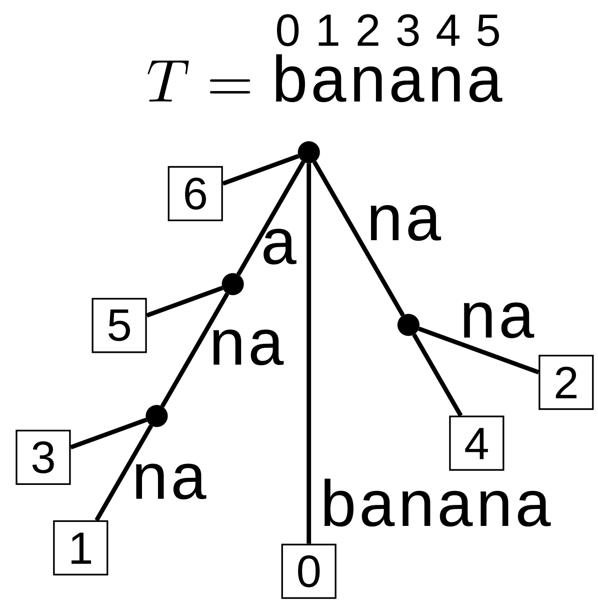
0	6	
1	5	a
2	3	ana
3	1	anana
4	0	banana
5	4	na
6	2	nana

Many of the applications need the longest common prefix array

- computable in linear time [Kasai et al., 2001]

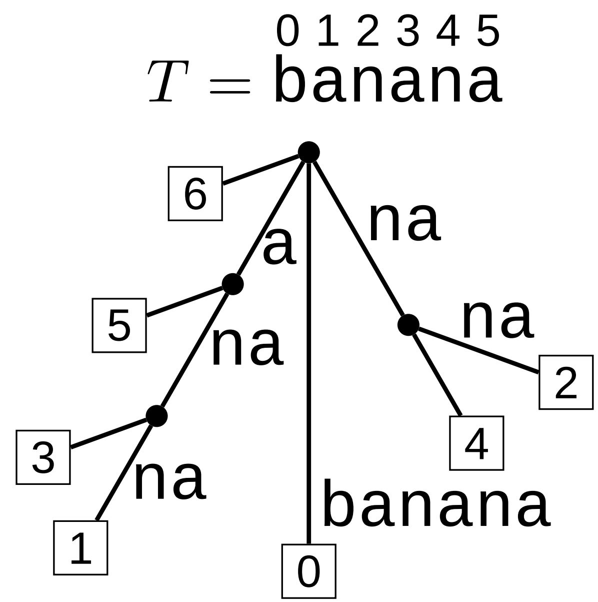
Suffix array vs. Suffix tree

- Suffix arrays are no more an inferior simplification of suffix trees
- many recent suffix array algorithms are
  - efficient in theory and practice
  - different from suffix tree algorithms
  - nontrivial, even surprising
- case in point: linear time construction



Suffix array vs. Suffix tree

- Suffix arrays are no more an inferior simplification of suffix trees
- many recent suffix array algorithms are
  - efficient in theory and practice
  - different from suffix tree algorithms
  - nontrivial, even surprising
- case in point: linear time construction



"In 2003 four papers have been published that collectively seem to establish the superiority of the suffix array over the suffix tree"

"Thus, if I were writing Chapter 5 today instead of in 2000/2001, I believe I would take a completely different approach: presenting suffix arrays as the main data structure"

— Bill Smyth: Errata on *Computing Patterns in Strings*

Alphabet

General alphabet

- only character comparisons in constant time
- lower bound $\Omega(n \log n)$ on suffix sorting

Constant alphabet

- constant number of distinct characters

Integer alphabet

- characters are integers from the range $[1, n]$

Alphabet

General alphabet

- only character comparisons in constant time
- lower bound $\Omega(n \log n)$ on suffix sorting

Constant alphabet

- constant number of distinct characters

Integer alphabet

- characters are integers from the range $[1, n]$
- order preserving renaming for other alphabets: sort characters and rename them with ranks
- linear time algorithm for integer alphabet

  $\Longrightarrow$ sorting suffixes is no harder than sorting characters

```
banana
  ⇓
 abn
  ⇓
 123
  ⇓
213131
```

6	
5	1
3	131
1	13131
0	213131
4	31
2	3131

History of linear time suffix array construction

1973 Suffix tree [Weiner]

- linear time construction for constant alphabet

1990 Suffix array [Manber & Myers]

- linear time construction only by conversion from suffix tree

1997 Integer alphabet [Farach]

- linear time suffix tree construction for integer alphabet

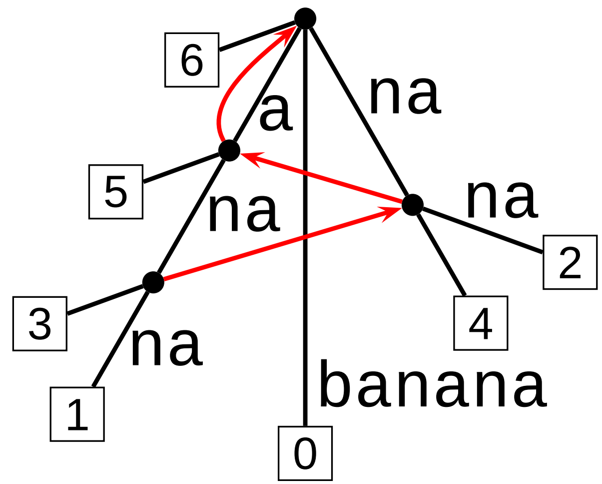
2003 Direct linear time suffix array construction [Ko & Aluru][Kim & al.][Kärkkäinen & Sanders]

- integer alphabet

Linear time suffix tree construction

- incremental algorithms
  [Weiner ’73] [McCreight ’76] [Ukkonen ’95]
  - add suffixes/characters one at a time
  - constant alphabet
  - suffix links needed
  - suffix automaton [Blumer et al., ’83]

$S =$ banana

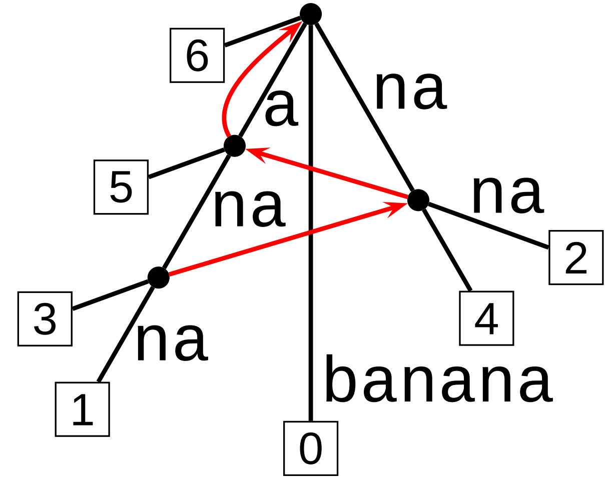


Linear time suffix tree construction

- divide-and-conquer [Farach '97]
  1. build suffix tree of $R = [t_0 t_1][t_2 t_3]\ldots$
  2. build odd and even tree
  3. merge them (complicated)

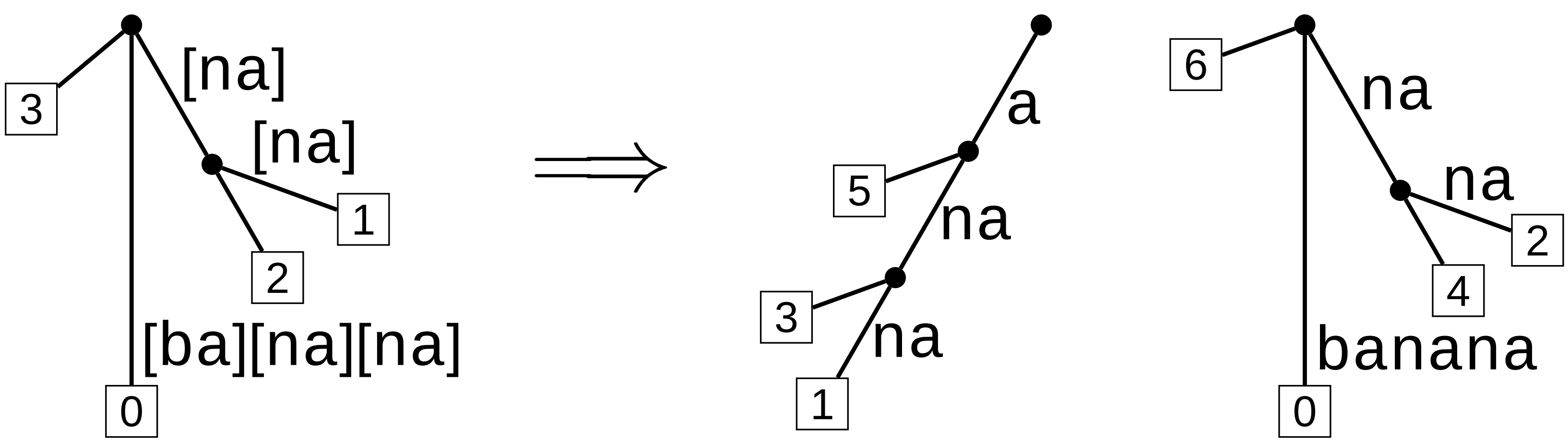
- integer alphabet
- suffix links needed in merging

$S =$ banana



$R =$ [ba][na][na]

odd tree even tree



Linear time suffix array construction

- three algorithms in June 2003

  A2: [Kim, Sim, Park & Park., CPM '03]

  A3: [Kärkkäinen & Sanders, ICALP '03]

  Ax: [Ko & Aluru, CPM '03]

- common structure: divide-and-conquer

  0. Choose a sample $\mathcal{S}$ of suffixes
  1. Sort the sample $\mathcal{S}$ by recursion
  2. Sort other suffixes $\bar{\mathcal{S}}$ using sorted $\mathcal{S}$
  3. Merge $\mathcal{S}$ and $\bar{\mathcal{S}}$

- rest of talk
  - step-by-step description

    Step 0 $\longrightarrow$ Step 3 ($\longrightarrow$ Step 1 $\longrightarrow$ Step 2)
  - all algorithms in parallel

Time complexity

0. Choose a sample $\mathcal{S}$ of suffixes
1. Sort the sample $\mathcal{S}$ by recursion
2. Sort other suffixes $\bar{\mathcal{S}}$ using sorted $\mathcal{S}$
3. Merge $\mathcal{S}$ and $\bar{\mathcal{S}}$

- integer alphabet
- excluding recursive call everything is linear
- recursion on text R over integer alphabet with $|R| = |\mathcal{S}| \leq 2n/3$
- time complexity $T(n) \leq \mathcal{O}(n) + T(2n/3) = \mathcal{O}(n)$

Step 0: Compute sample

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\} =$ odd suffixes [Kim & al.]

- sample size $n/2$

Step 0: Compute sample

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\} =$ odd suffixes [Kim & al.]

- sample size $n/2$

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \neq 0\} = \{S_1, S_2, S_4, S_5, S_7 \ldots\}$ [K & Sanders]

- sample size $2n/3$

Step 0: Compute sample

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\} =$ odd suffixes [Kim & al.]

- sample size $n/2$

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \neq 0\} = \{S_1, S_2, S_4, S_5, S_7 \ldots\}$ [K & Sanders]

- sample size $2n/3$

Ax: $\mathcal{S} =$ smaller of $\{S_i \mid S_i < S_{i+1}\}$ and $\{S_i \mid S_i > S_{i+1}\}$ [Ko & Aluru]

- sample size $\leq n/2$
- w.l.o.g. assume $\mathcal{S} = \{S_i \mid S_i < S_{i+1}\}$
- $S_i \in \mathcal{S} \quad \Longleftrightarrow \quad t_i < t_{i+1}$ or $t_i = t_{i+1}$ and $S_{i+1} \in \mathcal{S}$

Step 0: Compute sample: Example

$$S = \overset{0\,1\,2\,3\,4\,5}{\text{banana}}$$

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$

1	anana
3	ana
5	a

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \neq 0\}$

1	anana
2	nana
4	na
5	a

Ax: $\mathcal{S} = \{S_i \mid S_i < S_{i+1}\}$

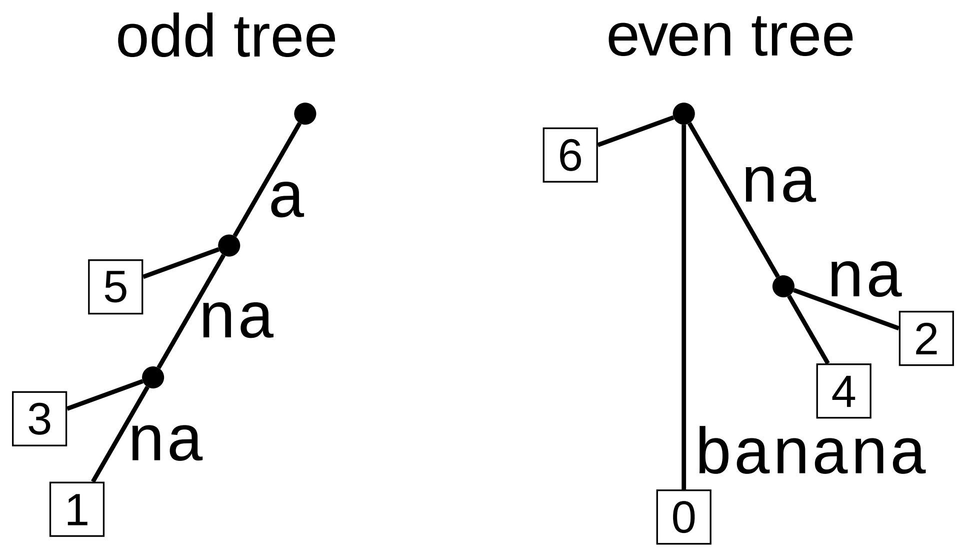
1	anana	$<$	nana
3	ana	$<$	na

banana	$>$	anana
nana	$>$	ana
na	$>$	a
a	$>$	

Step 3: Merge $\mathcal{S}$ and $\bar{\mathcal{S}}$

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$ $\bar{\mathcal{S}} = \{S_j \mid j \bmod 2 = 0\}$

- very complicated (simulates suffix tree?)



Step 3: Merge $\mathcal{S}$ and $\bar{\mathcal{S}}$

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$ $\quad\quad$ $\bar{\mathcal{S}} = \{S_j \mid j \bmod 2 = 0\}$

- very complicated (simulates suffix tree?)

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \neq 0\}$ $\quad\quad$ $\bar{\mathcal{S}} = \{S_j \mid j \bmod 3 = 0\}$

- standard comparison-based merge
- need to compare $S_i \in \mathcal{S}$ and $S_j \in \bar{\mathcal{S}}$:
- $i \bmod 3 = 1 \quad \Longrightarrow \quad S_{i+1}, S_{j+1} \in \mathcal{S}$
  $\Longrightarrow$ compare (t_i, S_{i+1}) and (t_j, S_{j+1})
- $i \bmod 3 = 2 \quad \Longrightarrow \quad S_{i+2}, S_{j+2} \in \mathcal{S}$
  $\Longrightarrow$ compare (t_i, t_{i+1}, S_{i+2}) and (t_j, t_{j+1}, S_{j+2})

Step 3: Merge $\mathcal{S}$ and $\bar{\mathcal{S}}$

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$ $\quad$ $\bar{\mathcal{S}} = \{S_j \mid j \bmod 2 = 0\}$

- very complicated (simulates suffix tree?)

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \neq 0\}$ $\quad$ $\bar{\mathcal{S}} = \{S_j \mid j \bmod 3 = 0\}$

- standard comparison-based merge
- need to compare $S_i \in \mathcal{S}$ and $S_j \in \bar{\mathcal{S}}$:
- $i \bmod 3 = 1 \quad \Longrightarrow \quad S_{i+1}, S_{j+1} \in \mathcal{S}$
  $\Longrightarrow$ compare (t_i, S_{i+1}) and (t_j, S_{j+1})
- $i \bmod 3 = 2 \quad \Longrightarrow \quad S_{i+2}, S_{j+2} \in \mathcal{S}$
  $\Longrightarrow$ compare (t_i, t_{i+1}, S_{i+2}) and (t_j, t_{j+1}, S_{j+2})

Ax: $\mathcal{S} = \{S_i \mid S_i < S_{i+1}\}$ $\quad$ $\bar{\mathcal{S}} = \{S_j \mid S_j > S_{j+1}\}$

- let $\mathcal{S}_c = \{S_i \in \mathcal{S} \mid t_i = c\}$ and $\bar{\mathcal{S}}_c = \{S_j \in \bar{\mathcal{S}} \mid t_j = c\}$
- suffix array is $\bar{\mathcal{S}}_a \mathcal{S}_a \bar{\mathcal{S}}_b \mathcal{S}_b \ldots$
- proof: $\bar{\mathcal{S}}_c \ni cab < ccc\ldots < cccd \in \mathcal{S}_c$

Merging in A2 and A3

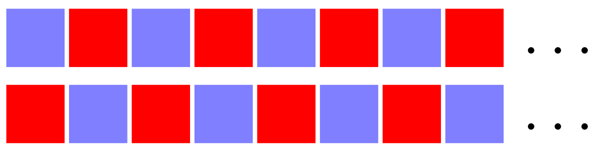
Problem: comparing sample and nonsample suffixes

= sample position = nonsample position

A2: Comparing odd and even suffixes

even

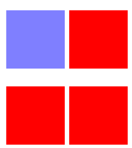
odd



A3: Comparing 0-suffixes and 1-suffixes

0-suffix

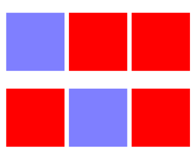
1-suffix



Comparing 0-suffixes and 2-suffixes

0-suffix

2-suffix



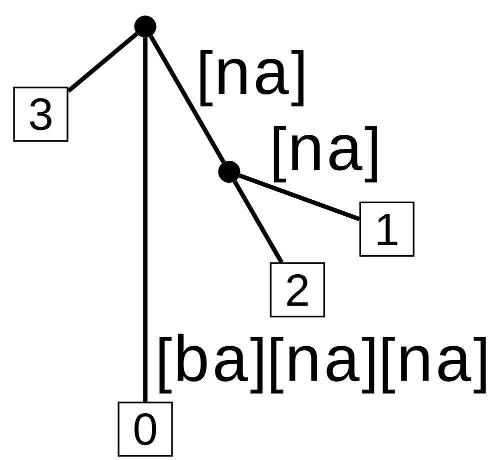
Step 1: Sort the sample

1. construct text R whose suffixes exactly represent sample $\mathcal{S}$
   - let $\mathcal{S} = \{S_{i_1}, S_{i_2}, S_{i_3}, \ldots\}$ with $i_1 < i_2 < i_3 < \cdots$
   - natural choice: $R = [t_{i_1} \ldots t_{i_2-1}][t_{i_2} \ldots t_{i_3-1}][t_{i_3} \ldots t_{i_4-1}] \ldots$
2. rename characters of R with ranks $\implies$ alphabet $[1, |R|]$
3. sort suffixes of R (recursion)

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$

- $R = [t_1 t_2][t_3 t_4] \ldots$

$R =$ [ba][na][na]



Step 1: Sort the sample

1. construct text R whose suffixes exactly represent sample $\mathcal{S}$
   - let $\mathcal{S} = \{S_{i_1}, S_{i_2}, S_{i_3}, \ldots\}$ with $i_1 < i_2 < i_3 < \cdots$
   - natural choice: $R = [t_{i_1} \ldots t_{i_2-1}][t_{i_2} \ldots t_{i_3-1}][t_{i_3} \ldots t_{i_4-1}] \ldots$
2. rename characters of R with ranks $\implies$ alphabet $[1, |R|]$

3. sort suffixes of R (recursion)

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$
- $R = [t_1 t_2][t_3 t_4] \ldots$

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \neq 0\}$
- $R \neq [t_1][t_2 t_3][t_4][t_5 t_6] \ldots$

Step 1: Sort the sample

1. construct text R whose suffixes exactly represent sample $\mathcal{S}$
   - let $\mathcal{S} = \{S_{i_1}, S_{i_2}, S_{i_3}, \ldots\}$ with $i_1 < i_2 < i_3 < \cdots$
   - natural choice: $R = [t_{i_1} \ldots t_{i_2-1}][t_{i_2} \ldots t_{i_3-1}][t_{i_3} \ldots t_{i_4-1}] \ldots$
2. rename characters of R with ranks $\Longrightarrow$ alphabet $[1, |R|]$
   - proper prefix problem: [a][a…] < [ab][…] < [a][c…]
3. sort suffixes of R (recursion)

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$

- $R = [t_1 t_2][t_3 t_4] \ldots$

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \neq 0\}$

- $R \neq [t_1][t_2 t_3][t_4][t_5 t_6] \ldots$

Step 1: Sort the sample

1. construct text R whose suffixes exactly represent sample $\mathcal{S}$
   - let $\mathcal{S} = \{S_{i_1}, S_{i_2}, S_{i_3}, \ldots\}$ with $i_1 < i_2 < i_3 < \cdots$
   - natural choice: $R = [t_{i_1} \ldots t_{i_2-1}][t_{i_2} \ldots t_{i_3-1}][t_{i_3} \ldots t_{i_4-1}] \ldots$
2. rename characters of R with ranks $\Longrightarrow$ alphabet $[1, |R|]$
   - proper prefix problem: $[\mathsf{a}][\mathsf{a} \ldots] < [\mathsf{ab}][\ldots] < [\mathsf{a}][\mathsf{c} \ldots]$
3. sort suffixes of R (recursion)

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$

- $R = [t_1 t_2][t_3 t_4] \ldots$

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \neq 0\}$

- $R = [t_1 t_2 t_3][t_4 t_5 t_6] \ldots [t_2 t_3 t_4][t_5 t_6 t_7] \ldots$

Step 1: Sort the sample

1. construct text R whose suffixes exactly represent sample $\mathcal{S}$
   - let $\mathcal{S} = \{S_{i_1}, S_{i_2}, S_{i_3}, \ldots\}$ with $i_1 < i_2 < i_3 < \cdots$
   - natural choice: $R = [t_{i_1} \ldots t_{i_2-1}][t_{i_2} \ldots t_{i_3-1}][t_{i_3} \ldots t_{i_4-1}] \ldots$
2. rename characters of R with ranks $\Longrightarrow$ alphabet $[1, |R|]$
   - proper prefix problem: [a][a$\ldots$] $<$ [ab][$\ldots$] $<$ [a][c$\ldots$]
3. sort suffixes of R (recursion)

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$
   - $R = [t_1 t_2][t_3 t_4] \ldots$

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \neq 0\}$
   - $R = [t_1 t_2 t_3][t_4 t_5 t_6] \ldots [t_2 t_3 t_4][t_5 t_6 t_7] \ldots$

Ax: $\mathcal{S} = \{S_i \mid S_i < S_{i+1}\}$
   - $R = [t_{i_1} \ldots t_{i_2-1} t_{i_2} \infty][t_{i_2} \ldots t_{i_3-1} t_{i_3} \infty][t_{i_3} \ldots t_{i_4-1} t_{i_4} \infty] \ldots$

Step 1: Sort the sample: Example

$$\begin{array}{l} \text{0 1 2 3 4 5} \\ S = \text{banana} \end{array}$$

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$

1	anana	$R =$ [an][an][a]
3	ana	[an][a]
5	a	[a]

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \neq 0\}$

1	anana	$R =$ [ana][na][nan][a]
2	nana	[nan][a]
4	na	[na][nan][a]
5	a	[a]

Ax: $\mathcal{S} = \{S_i \mid S_i < S_{i+1}\}$

1	anana	$R =$ [ana∞][ana]
3	ana	[ana]

Step 2: Sort other suffixes $\bar{\mathcal{S}}$

- Let $next(\bar{\mathcal{S}}) = \{S_{j+1} \mid S_j \in \bar{\mathcal{S}}\}$ and $\bar{\mathcal{S}}_c = \{S_j \in \bar{\mathcal{S}} \mid t_j = c\}$
- For each $S_i \in next(\bar{\mathcal{S}})$ in sorted order
  insert S_{i-1} into $\bar{\mathcal{S}}_c$ with $c = t_{i-1}$

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$ $\qquad$ $\bar{\mathcal{S}} = \{S_j \mid j \bmod 2 = 0\}$

- $next(\bar{\mathcal{S}}) = \mathcal{S}$

Step 2: Sort other suffixes $\bar{\mathcal{S}}$

- Let $next(\bar{\mathcal{S}}) = \{S_{j+1} \mid S_j \in \bar{\mathcal{S}}\}$ and $\bar{\mathcal{S}}_c = \{S_j \in \bar{\mathcal{S}} \mid t_j = c\}$
- For each $S_i \in next(\bar{\mathcal{S}})$ in sorted order
  insert S_{i-1} into $\bar{\mathcal{S}}_c$ with $c = t_{i-1}$

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$ $\quad\quad$ $\bar{\mathcal{S}} = \{S_j \mid j \bmod 2 = 0\}$

- $next(\bar{\mathcal{S}}) = \mathcal{S}$

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \neq 0\}$ $\quad\quad$ $\bar{\mathcal{S}} = \{S_j \mid j \bmod 3 = 0\}$

- $next(\bar{\mathcal{S}}) \subset \mathcal{S}$

Step 2: Sort other suffixes $\bar{\mathcal{S}}$

- Let $next(\bar{\mathcal{S}}) = \{S_{j+1} \mid S_j \in \bar{\mathcal{S}}\}$ and $\bar{\mathcal{S}}_c = \{S_j \in \bar{\mathcal{S}} \mid t_j = c\}$
- For each $S_i \in next(\bar{\mathcal{S}})$ in sorted order
  insert S_{i-1} into $\bar{\mathcal{S}}_c$ with $c = t_{i-1}$

A2: $\mathcal{S} = \{S_i \mid i \bmod 2 \neq 0\}$ $\quad$ $\bar{\mathcal{S}} = \{S_j \mid j \bmod 2 = 0\}$

- $next(\bar{\mathcal{S}}) = \mathcal{S}$

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \neq 0\}$ $\quad$ $\bar{\mathcal{S}} = \{S_j \mid j \bmod 3 = 0\}$

- $next(\bar{\mathcal{S}}) \subset \mathcal{S}$

Ax: $\mathcal{S} = \{S_i \mid S_i < S_{i+1}\}$ $\quad$ $\bar{\mathcal{S}} = \{S_j \mid S_j > S_{j+1}\}$

- scan suffix array $\epsilon \bar{\mathcal{S}}_a \mathcal{S}_a \bar{\mathcal{S}}_b \mathcal{S}_b \ldots$
- if suffix S_i is in $next(\bar{\mathcal{S}})$ insert S_{i-1}
- when scan reaches $S_j \in \bar{\mathcal{S}}$ it is already in place
  because $S_{j+1} < S_j$

Implementating A3: Subroutines

```
// compare pairs and triples
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)); }

// radix sort (one pass)
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 occurrences
   for (int i = 0, sum = 0; i <= K; i++)          // exclusive prefix sums
   { int t = c[i]; c[i] = sum; sum += t; }
   // sort
   for (int i = 0; i < n; i++) b[c[r[a[i]]]++] = a[i];
   delete [] c;
}
```

Implementating A3: Main function

```
// compute suffix array of s
// require s[n]=s[n+1]=s[n+2]=0, n>=2
void suffixArray(int* s, int* SA, int n, int K) {

   // initialize
   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];
```

Step 0: Compute sample
Step 1: Sort sample
Step 2: Sort other suffi xes
Step 3: Merge

```
   // clean up
   delete [] s12; delete [] SA12; delete [] SA0; delete [] s0;
}
```

Implementing A3: Step 0: Compute sample

```
// compute sample
for (int i=0, j=0; i < n+(n0-n1); i++) if (i%3 != 0) s12[j++] = i;
```

Implementing A3: Step 1: Sort the sample

```
// sort supercharacters (triples)
radixPass(s12 , SA12, s+2, n02, K);
radixPass(SA12, s12 , s+1, n02, K);
radixPass(s12 , SA12, s , n02, K);

// construct recursive text
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; }   // first half
   else { s12[SA12[i]/3 + n0] = name; }                // second half
}

if (name < n02) { // recurse if all supercharacters are not unique
   suffixArray(s12, SA12, n02, name);
   for (int i = 0; i < n02; i++) s12[SA12[i]] = i + 1;
} else   // end of recursion:  supercharacters are all unique
   for (int i = 0; i < n02; i++) SA12[s12[i] - 1] = i;
```

Implementing A3: Step 2: Sort other suffixes

```
// construct nonsample in order of next(nonsample)
for (int i=0, j=0; i < n02; i++)
   if (SA12[i] < n0) s0[j++] = 3*SA12[i];
// sort stably by first character
radixPass(s0, SA0, s, n0, K);
```

Implementing A3: Step 3: Merge

```
   // merge sample and nonsample suffixes
   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();
      int j = SA0[p];
      if (SA12[t] < n0 ?  // compare
         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]))
      {          // sample suffix is smaller
         SA[k] = i; t++;
         if (t == n02)   // done --- only nonsample suffixes left
            for (k++; p < n0; p++, k++) SA[k] = SA0[p];
      } else {  // nonsample suffix is smaller
         SA[k] = j; p++;
         if (p == n0)    // done --- only sample suffixes left
            for (k++; t < n02; t++, k++) SA[k] = GetI();
      }
   }
```

Concluding remarks

- Implementation
  - **A3** and **Ax** are practical algorithms
  - can be made space-efficient
- Other models of computation
  - **A3** is easily parallelizable and externalizable
  - improved BSP and EREW-PRAM algorithms [K & Sanders, '03]
  - fast external memory implementation [Dementiev & al, '05]
- Related construction algorithms
  - $\mathcal{O}(vn + n\log n)$ time, $\mathcal{O}(n/\sqrt{v})$ extra space ($v \in [3, n]$)
    fast and space-efficient in practice [Burkhardt & K, '03]
  - $\mathcal{O}(vn)$ time, $\mathcal{O}(n/\sqrt{v})$ extra space [K & Sanders]

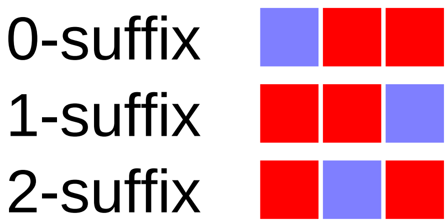
Open problems

- Suffix array has emerged from the shadow of suffix tree
  - several recent algoritms
  - missing algorithms?
- I still don't understand suffix arrays!
  - surprising algorithms
  - common combinatorial principles?
  - more surprises coming?

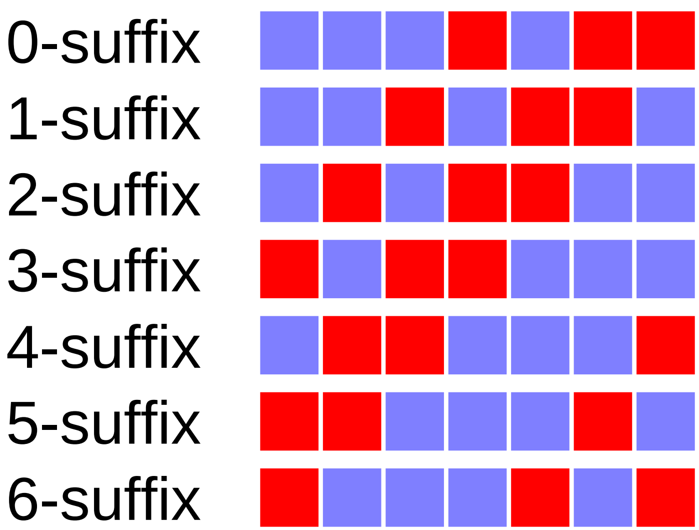
Difference cover samples

= sample position = nonsample position

A3: $\mathcal{S} = \{S_i \mid i \bmod 3 \in \{1,2\}\}$



A7: $\mathcal{S} = \{S_i \mid i \bmod 7 \in \{3,5,6\}\}$



Difference cover samples

$D \subseteq [0, v)$ is a difference cover modulo v if

$$\{i - j \bmod v \mid i, j \in D\} = [0, v)$$

- $D = \{1, 2\}$ is a difference cover modulo 3
- $D = \{3, 5, 6\}$ is a difference cover modulo 7
- $D = \{1\}$ is not a difference cover modulo 2

Algorithms

- **A3**
- $\mathcal{O}(vn + n \log n)$ time, $\mathcal{O}(n/\sqrt{v})$ extra space [Burkhardt & K, '03]
- $\mathcal{O}(vn)$ time, $\mathcal{O}(n/\sqrt{v})$ extra space [K & Sanders, ??]