Suffix Sorting by Difference Cover Sampling

Juha Kärkkäinen

University of Helsinki

Summer School on Algorithmic Data Analysis (SADA07)

Helsinki, 01 June 2007

Suffix Sorting

Given a string T

$$T = \text{BANANA}$$

sort the **suffixes** of T

BANANA		
ANANA		A
NANA		ANA
ANA	$\longrightarrow$	ANANA
NA		BANANA
A		NA
		NANA

Outline

1. Suffix sorting
2. Two techniques
3. Linear-time and I/O-optimal suffix sorting
4. Difference cover sampling
5. Space efficient Burrows-Wheeler transform

Outline

1. **Suffix sorting**
   - Problem
   - Applications
   - Solutions
2. Two techniques
3. Linear-time and I/O-optimal suffix sorting
4. Difference cover sampling
5. Space efficient Burrows-Wheeler transform

Suffix Sorting

Given a string T of length n over alphabet Σ

$$T = \text{BANANA}$$

sort the suffixes of T

BANANA		
ANANA		A
NANA		ANA
ANA	$\longrightarrow$	ANANA
NA		BANANA
A		NA
		NANA

Suffix Sorting

Given a string T of length n over alphabet Σ

$$T = \text{BANANA\#}$$

sort the suffixes of T $\# < x$ for all $x \in \Sigma$

BANANA#		#
ANANA#		A#
NANA#		ANA#
ANA#	$\longrightarrow$	ANANA#
NA#		BANANA#
A#		NA#
#		NANA#

Suffix Array

$$T \quad = \quad \overset{0\,1\,2\,3\,4\,5\,6}{\text{BANANA\#}}$$

0	BANANA#
1	ANANA#
2	NANA#
3	ANA#
4	NA#
5	A#
6	#

$\longrightarrow$

6	#
5	A#
3	ANA#
1	ANANA#
0	BANANA#
4	NA#
2	NANA#

Suffix array

Burrows–Wheeler Transform (BWT)

$$T = \begin{matrix} \text{0 1 2 3 4 5 6} \\ \text{BANANA\#} \end{matrix}$$

0	BANANA#	$\longrightarrow$	6	#BANANA
1	ANANA#B		5	A#BANAN
2	NANA#BA		3	ANA#BAN
3	ANA#BAN		1	ANANA#B
4	NA#BANA		0	BANANA#
5	A#BANAN		4	NA#BANA
6	#BANANA		2	NANA#BA

Burrows–Wheeler Transform (BWT)

$$T = \underset{0\,1\,2\,3\,4\,5\,6}{\text{BANANA\#}}$$

0	BANANA	#
1	ANANA#	B
2	NANA#B	A
3	ANA#BA	N
4	NA#BAN	A
5	A#BANA	N
6	#BANAN	A

$\longrightarrow$

6	#BANAN	A
5	A#BANA	N
3	ANA#BA	N
1	ANANA#	B
0	BANANA	#
4	NA#BAN	A
2	NANA#B	A

BWT

Applications

- Suffix array is a full-text index
  - searching
  - text mining
- Construction of other text indexes
  - suffix tree
  - string B-tree
  - compressed indexes
- Text compression (BWT)

$T =$ BANANA#

6	#
5	A#
3	ANA#
1	ANANA#
0	BANANA#
4	NA#
2	NANA#

$BWT =$ ANNB#AA

Suffix sorting is usually the computational bottleneck

Solutions

General sorting

- single comparison can take $\Omega(n)$ time
- $\Omega(n^2 \log n)$ time

String sorting

- total length of suffixes is $\Theta(n^2)$
- $\Omega(n^2)$ time

Suffix sorting

- many algoritms with $\mathcal{O}(n \log n)$ or $\mathcal{O}(n)$ time

AAA...AA#

#
A#
AA#
AAA#
:
:
AAA...AA#

Suffix Sorting Algorithms

Most algorithms do random accesses to the text

- Not I/O-efficient

Most algorithms use an array of n integers

- Not always space efficient
  - BWT computation

    $$|T| = |\text{BWT}| = \mathcal{O}(n \log |\Sigma|) \text{ bits or}$$
    $$nH_k(T) + o(n \log |\Sigma|) \text{ bits}$$
    $$|\text{array of integers}| = \Omega(n \log n) \text{ bits}$$
  - sorting a subset of suffixes

Outline

1. Suffix sorting
2. **Two techniques**
   - Sorting Periodic Subset
   - Induced Sorting
3. Linear-time and I/O-optimal suffix sorting
4. Difference cover sampling
5. Space efficient Burrows-Wheeler transform

Sorting Periodic Subsets

- Sort odd suffixes

1	ANANA#
3	ANA#
5	A#

$\longrightarrow$

5	A#
3	ANA#
1	ANANA#

Sorting Periodic Subsets

- Sort odd suffixes

1	ANANA#	[AN][AN][A#]	221
3	ANA#	[AN][A#]	21
5	A#	[A#]	1

$\longrightarrow$

5	A#	[A#]	1
3	ANA#	[AN][A#]	21
1	ANANA#	[AN][AN][A#]	221

- Reduction to sorting all suffixes of a string of length $n/2$

(B)ANANA# $\longrightarrow$ [AN][AN][A#] $\longrightarrow$ 221

Sorting Periodic Subsets

- Sort 2 out of 3 suffixes $i \bmod 3 \in \{1, 2\}$

1	ANANA#
2	NANA#
4	NA#
5	A#

$\longrightarrow$

5	A#
1	ANANA#
4	NA#
2	NANA#

Sorting Periodic Subsets

- Sort 2 out of 3 suffixes $i \bmod 3 \in \{1, 2\}$

1	ANANA#	[ANA][NA#][NAN][A##]	2341
2	NANA#	[NA#][NAN][A##]	341
4	NA#	[NAN][A##]	41
5	A#	[A##]	1

$\longrightarrow$

5	A#	[A##]	1
1	ANANA#	[ANA][NA#][NAN][A##]	2341
4	NA#	[NA#][NAN][A##]	341
2	NANA#	[NAN][A##]	41

- Reduction to sorting all suffixes of a string of length $2n/3$

(B)ANANA#(BA)NANA## $\rightarrow$ [ANA][NA#][NAN][A##] $\rightarrow$ 2341

Sorting Periodic Subsets

- Sort d out of v suffixes $\qquad i \bmod v \in D$, $|D| = d$
- Reduction to sorting all suffixes of a string of length dn/v
- Reduction requires sorting dn/v strings of length v
  - time: $\mathcal{O}((dn/v)\log(dn/v) + dn)$ or $\mathcal{O}(dn)$
  - space: $\mathcal{O}(dn/v + v)$ integers

Sorting Periodic Subsets

- Sort d out of v suffixes $\quad i \bmod v \in D$, $|D| = d$
- Reduction to sorting all suffixes of a string of length dn/v
- Reduction requires sorting dn/v strings of length v
  - time: $\mathcal{O}((dn/v)\log(dn/v) + dn)$ or $\mathcal{O}(dn)$
  - space: $\mathcal{O}(dn/v + v)$ integers

- For constant v
  - $\mathcal{O}(n \log n)$ or $\mathcal{O}(n)$ time
  - $\mathcal{O}\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$ I/Os in external memory

Sorting time

Induced Sorting

Sort even suffixes given sorted odd suffixes

5	A#
3	ANA#
1	ANANA#

$\longrightarrow$

6	#
5-1=4	NA#
3-1=2	NANA#
1-1=0	BANANA#

$\longrightarrow$

6	#
0	BANANA#
4	NA#
2	NANA#

Induced Sorting

Sort even suffixes given sorted odd suffixes

5	A#
3	ANA#
1	ANANA#

$\longrightarrow$

6	#
5-1=4	NA#
3-1=2	NANA#
1-1=0	BANANA#

$\longrightarrow$

6	#
0	BANANA#
4	NA#
2	NANA#

Sort $i \bmod 3 = 0$-suffixes using $i \bmod 3 = 1$-suffixes

1	ANANA#
4	NA#

$\longrightarrow$

6	#
1-1=0	BANANA#
4-1=3	ANA#

$\longrightarrow$

6	#
3	ANA#
0	BANANA#

Induced Sorting

Sort even suffixes given sorted odd suffixes

5	A#
3	ANA#
1	ANANA#

$\longrightarrow$

6	#
5-1=4	NA#
3-1=2	NANA#
1-1=0	BANANA#

$\longrightarrow$

6	#
0	BANANA#
4	NA#
2	NANA#

Sort $i \bmod 3 = 0$-suffixes using $i \bmod 3 = 1$-suffixes

1	ANANA#
4	NA#

$\longrightarrow$

6	#
1-1=0	BANANA#
4-1=3	ANA#

$\longrightarrow$

6	#
3	ANA#
0	BANANA#

- Sorting time

Outline

1. Suffix sorting
2. Two techniques
3. **Linear-time and I/O-optimal suffix sorting**
   - Algorithm
   - Analysis
   - Experiments
4. Difference cover sampling
5. Space efficient Burrows-Wheeler transform

Linear-Time and I/O-Optimal Suffix Sorting

[Farach, FOCS '97]
[Farach–Colton & Ferragina & Muthukhrisnan, JACM '00]

1. Sort odd suffixes
   - reduction to suffix sorting on string of length $n/2$
   - **recurse** for reduced problem
2. Sort even suffixes
   - induced sorting from odd suffixes
3. Merge
   - very complicated

Linear-Time and I/O-Optimal Suffix Sorting

[K & Sanders, ICALP '03]
[K & Sanders & Burkhardt, JACM '06]

1. Sort $i \bmod 3 \in \{1, 2\}$-suffixes
   - reduction to suffix sorting on string of length $2n/3$
   - **recurse** for reduced problem
2. Sort $i \bmod 3 = 0$-suffixes
   - induced sorting from $i \bmod 3 = 1$-suffixes
3. Merge
   - simple comparison-based merging
   - uses constant-time suffix comparisons

Constant-Time Suffix Comparisons

- Assign ranks for $i \bmod 3 \in \{1, 2\}$-suffixes

5	A#	1
1	ANANA#	2
4	NA#	3
2	NANA#	4

- comparing $i \bmod 3 = 0$-suffix and $i \bmod 3 = 1$-suffix

3	ANA#	A$[$NA#$]$	A3
1	ANANA#	A$[$NANA#$]$	A4

- comparing $i \bmod 3 = 0$-suffix and $i \bmod 3 = 2$-suffix

3	ANA#	AN$[$A#$]$	AN1
2	NANA#	NA$[$NA#$]$	NA3

Analysis

- All except recursive call in sorting time
  - $\mathcal{O}(n \log n)$ or $\mathcal{O}(n)$ time
  - $\mathcal{O}\left(\frac{n}{B} \log_{M/B} \frac{n}{B}\right)$ I/Os in external memory
- Recursive call does not increase complexity

$$f(n) = g(n) + f(\alpha n) \quad \Longrightarrow \quad f(n) = \Theta(g(n))$$

given $\alpha < 1$ and $g(n) = \Omega(n)$

Thus

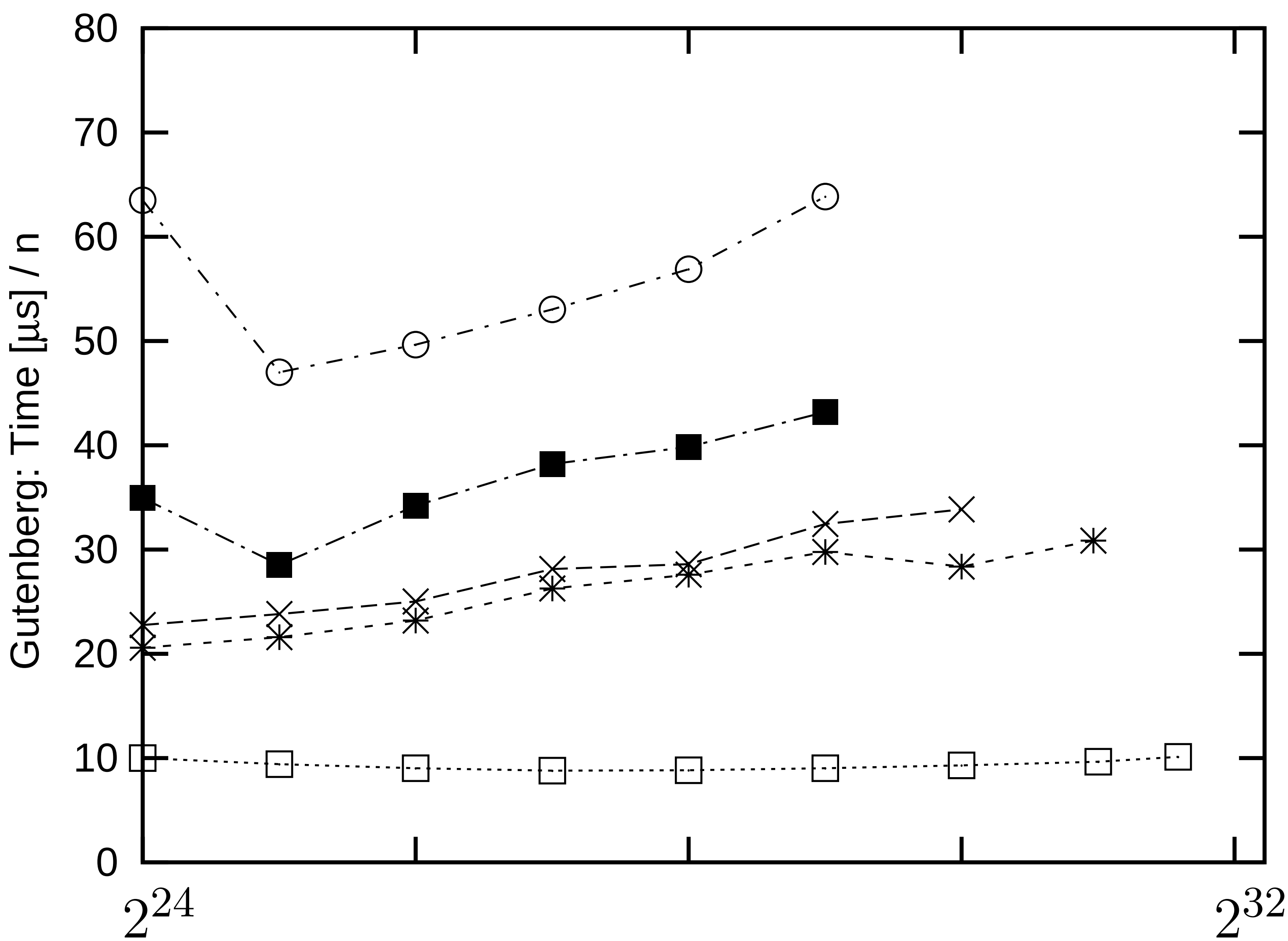
- Linear-time suffix sorting for integer alphabet
- External memory suffix sorting in $\mathcal{O}\left(\frac{n}{B} \log_{M/B} \frac{n}{B}\right)$ I/Os

Implementation of Linear-Time Algorithm

```
inline bool leq(int a1, int a2, int b1, int b2)  // lexicographic order
{ return(a1 < b1 || a1 == b1 && a2 <= b2); }     // for pairs
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));  // and triples

// 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 occurrences
    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;
}

// find the suffix array SA of s[0..n-1] in 1..K^n
// require s[n]=s[n+1]=s[n+2]=0, n>=2
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);

    // 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
    }

    // 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;

    // stably sort the mod 0 suffixes from SA12 by their first character
    for (int i=0, j=0; i < n02; i++) if (SA12[i] < n0) s0[j++] = 3*SA12[i];
    radixPass(s0, SA0, s, n0, K);

    // merge sorted SA0 suffixes and sorted SA12 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();     // pos of current offset 12 suffix
        int j = SA0[p];     // pos of current offset 0 suffix
        if (SA12[t] < n0 ?  // different compares for mod 1 and mod 2 suffixes
            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 {  // suffix from SA0 is smaller
            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;
}
```

Experiments with External Memory Algorithm

[Dementiev & K & Mehnert & Sanders, Alenex ’05, JEA ’07]



Outline

1. Suffix sorting
2. Two techniques
3. Linear-time and I/O-optimal suffix sorting
4. **Difference cover sampling**
    - Overview
    - Definition
    - Usage
5. Space efficient Burrows-Wheeler transform

Difference Cover Sampling

Data structure $DCS_v(T)$ [Burkhardt & K, CPM '03]

- computed from text T of length n
- parameter $v \in [3, n^{2/3})$

Properties

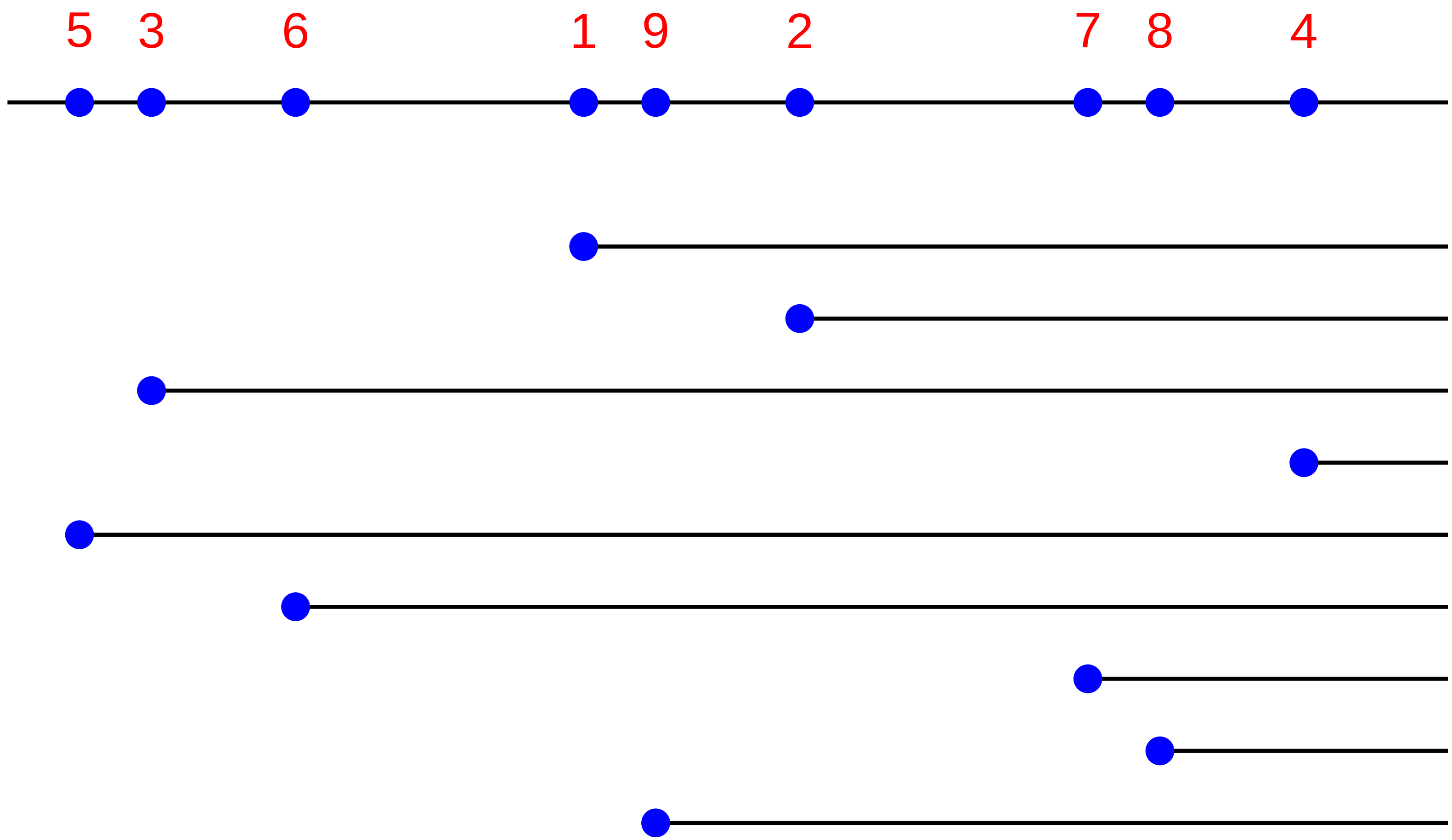
- small size
- fast(?) construction

Supports

- fast comparison of suffixes
- compact representation of suffixes

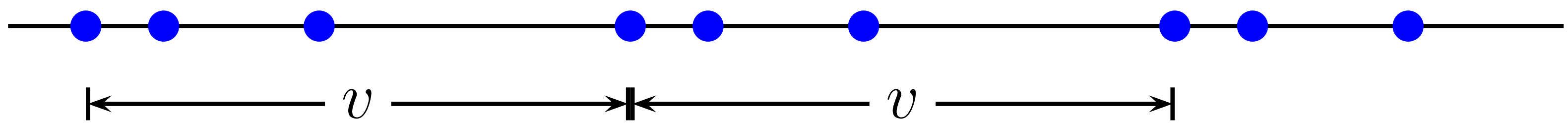
Basic Idea

- Sort a **sample** of suffixes and store the **ranks**

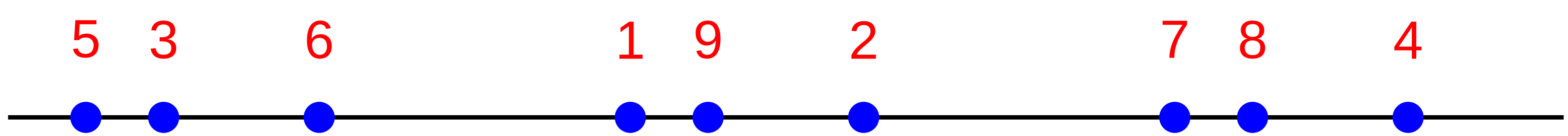


The Sample

The sample is periodic with period length v



- fast construction
- compact storage of ranks



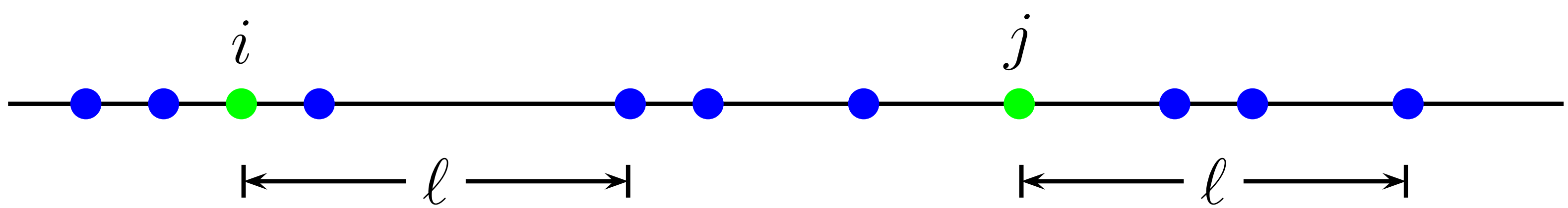
5	3	6	1	9	2	7	8	4

- needs lookup table of size v

The Sample

The period is a difference cover of size $\mathcal{O}(\sqrt{v})$

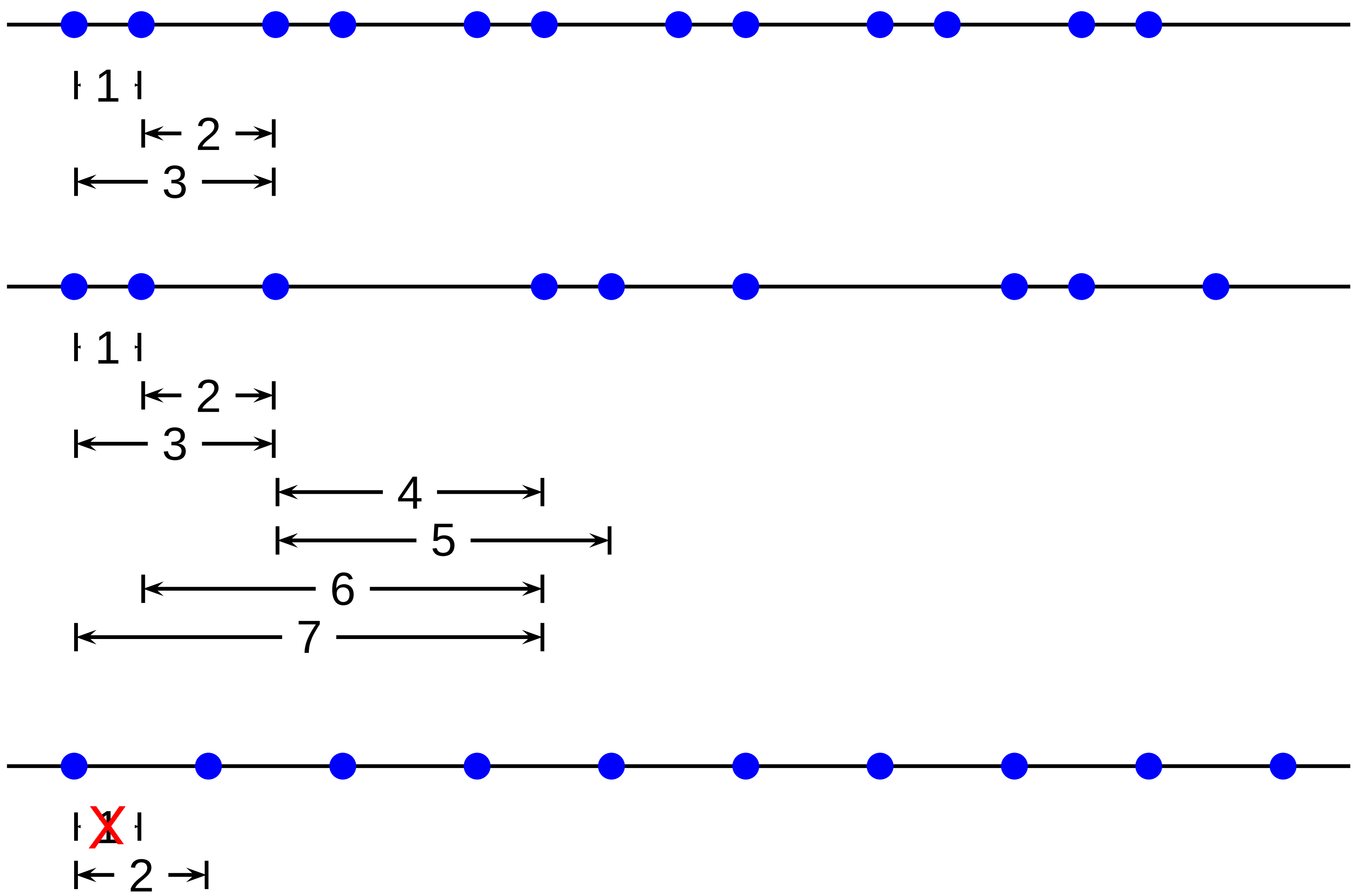
- For any i, j, there is $\ell \in [0, v)$ such that $i + \ell$ and $j + \ell$ are both sample positions



- Easy to compute [Colbourn & Ling '00]
- Sample size $\mathcal{O}(n/\sqrt{v})$

Difference cover

Covers all distances (differences)



Difference Cover Sampling

Data structure $DCS_v(T)$

Properties

- small size: $\mathcal{O}(n/\sqrt{v})$ integers
- fast(?) construction: $\mathcal{O}((n/\sqrt{v})\log(n/\sqrt{v}) + \sqrt{v}n)$ or $\mathcal{O}(\sqrt{v}n)$

Supports

- fast comparison of suffixes
- compact representation of suffixes

Difference Cover Sampling

Data structure $DCS_v(T)$

Properties

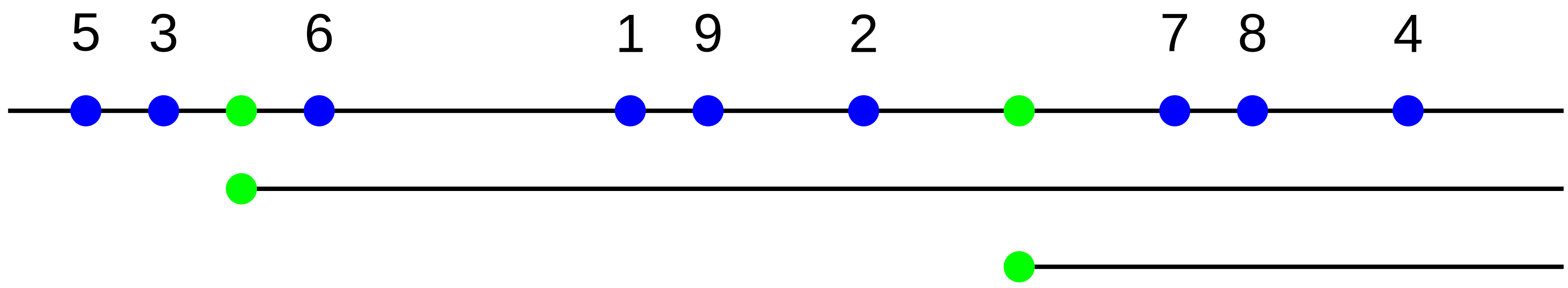
- small size: $\mathcal{O}(n/\sqrt{v})$ integers
- fast(?) construction: $\mathcal{O}((n/\sqrt{v})\log(n/\sqrt{v}) + \sqrt{v}n)$ or $\mathcal{O}(\sqrt{v}n)$

Supports

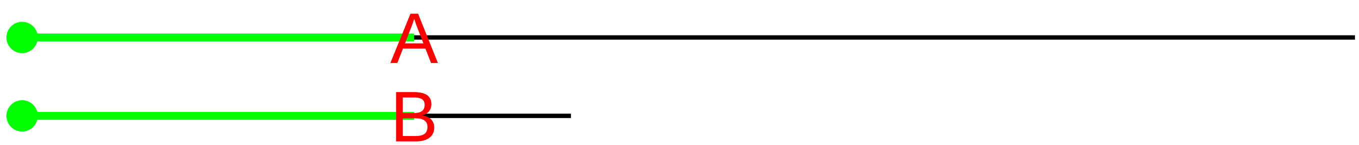
- fast comparison of suffixes
- compact representation of suffixes

Basic Idea

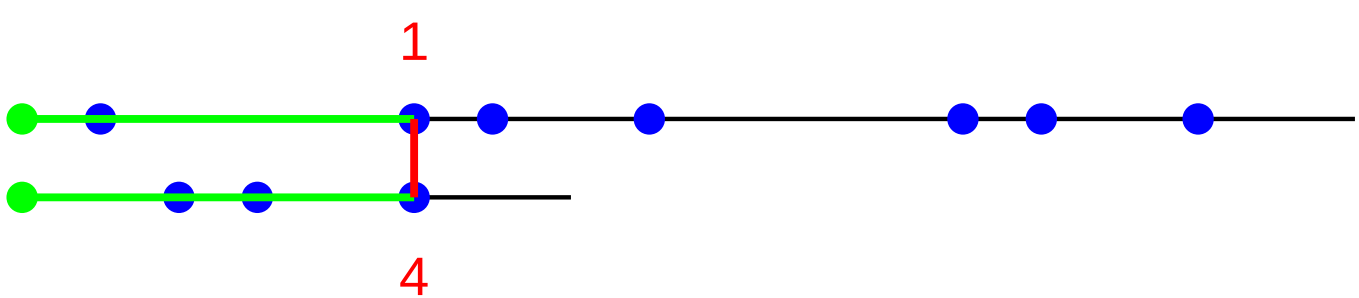
The order of two suffixes is determined ...



by first mismatch

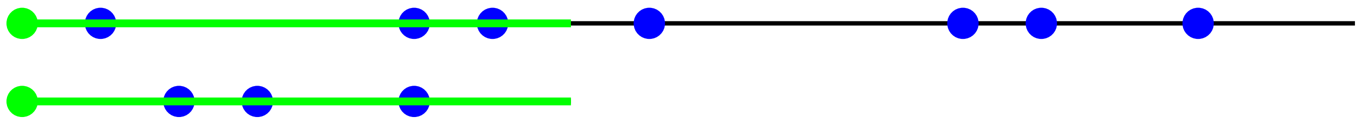


or by aligned sample points

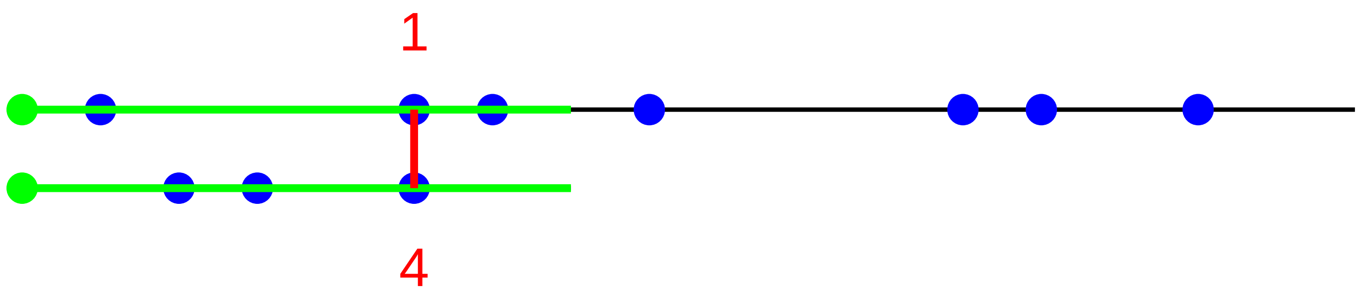


Fast Comparison of Suffixes

Given two suffixes with common prefix of length v

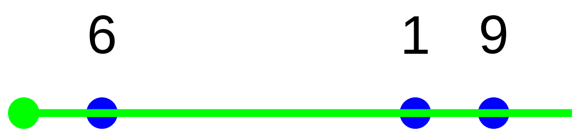


find aligned sample points within common prefix



- can be done in constant time
  using lookup table of size v

Compact Representation of Suffixes



- v characters
- $\mathcal{O}(\sqrt{v})$ ranks
- $\mathcal{O}(v)$ time comparison

Difference Cover Sampling

Properties

- small size: $\mathcal{O}(n/\sqrt{v})$ integers
- fast(?) construction: $\mathcal{O}((n/\sqrt{v})\log(n/\sqrt{v}) + \sqrt{v}n)$ or $\mathcal{O}(\sqrt{v}n)$

Supports

- fast comparison of suffixes
  - constant time comparison of two suffixes with a common prefix of length v
- compact representation of suffixes
  - v characters and $\mathcal{O}(\sqrt{v})$ integers
  - $\mathcal{O}(v)$ time comparison

Difference Cover Sampling

Set $v = 3$

Properties

- small(?) size: $\mathcal{O}(n)$ integers
- fast construction: sorting time

Supports

- fast comparison of suffixes
  - constant time comparison of two suffixes
- compact representation of suffixes
  - 2 characters and 2 integers
  - constant time comparison

Outline

1. Suffix sorting
2. Two techniques
3. Linear-time and I/O-optimal suffix sorting
4. Difference cover sampling
5. **Space efficient Burrows-Wheeler transform**
   - Overview
   - Definition
   - Usage

Space-Efficient BWT

Problem

- Most suffix sorting algorithm need an array of n integers

  $|\text{array of integers}| = \Omega(n \log n)$ bits

- T and its BWT may be much smaller

  $|T| = |\text{BWT}| = \mathcal{O}(n \log |\Sigma|)$ bits or

  $nH_k(T) + o(n \log |\Sigma|)$ bits

Solution: **Blockwise suffix sorting**

Burrows–Wheeler Transform (BWT)

0 1 2 3 4 5 6

T = BANANA#

6	#BANAN	A
5	A#BANA	N
3	ANA#BA	N
1	ANANA#	B
0	BANANA	#
4	NA#BAN	A
2	NANA#B	A

BWT

BWT by Blockwise Suffix Sorting

0 1 2 3 4 5 6

T = BANANA#

6	#BANAN	A
5	A#BANA	N
3	ANA#BA	N

BWT = ANN

BWT by Blockwise Suffix Sorting

0 1 2 3 4 5 6

T = BANANA#

		A
		N
3	ANA#BA	N
1	ANANA#	B
0	BANANA	#

BWT = ANNB$

BWT by Blockwise Suffix Sorting

0 1 2 3 4 5 6

T = BANANA#

		A
		N
		N
		B
		#
4	NA#BAN	A
2	NANA#B	A

BWT = ANNB$AA

Blockwise Suffix Sorting

[K, TCS '07?]

1. Choose a (random) set of suffixes as splitters to divide the suffixes into blocks
2. For each block, collect and sort the suffixes

- Key subproblems:
  - sorting splitters and blocks
  - collecting suffixes
- Using DCS_v
  - time $\mathcal{O}(n \log n + nv)$
  - space $\mathcal{O}(n/\sqrt{v})$

Experiments: Computing BWT

Runtime (in seconds) and memory footprint (in GBytes)

	text size = 256 MB				text size = 1 GB	
text	DCS	dnaDCS	MF	BK	DCS	dnaDCS
english	546	–	287	573	2746	–
rand-64	511	–	241	605	2566	–
repeat-64	2994	–	43751	1372	12779	–
DNA	585	1974	233	589	–	–
rand-DNA	574	1876	237	582	2898	10771
repeat-DNA	2986	12619	70125	1323	12555	52668
memory	0.46	0.23	1.3	1.5	1.8	0.90

Summary

Difference Cover Sampling

- flexible technique for suffix sorting

Applications

- simple linear-time suffix sorting
- I/O-efficient suffix sorting
- space-efficient BWT
- fast and space-efficient suffix array construction
  [Burkhardt & K, CPM '03]
- parallel suffix sorting [Kulla & Sanders, EuroPVM/MPI '06]