Cache-efficient string sorting for Burrows-Wheeler Transform

Advait D. Karande

Sriram Saroop

What is Burrows-Wheeler Transform?

- A pre-processing step for data compression
- Involves sorting of all rotations of the block of data to be compressed
- Rationale: Transformed string compresses better

Burrows-Wheeler Transform

s : abraca → s′ : caraab, I=1

0	aabrac
1	abraca
2	acaabr
3	bracaa
4	caabra
5	racaab

Suffix sorting for BWT

- Suffix array construction
- Effect of block size, N
  - Higher N -> Better compression

    -> Slower SA construction
- Suffix array construction algorithms
  - Quick sort based $O(N^2(\log N))$: Used in bzip2
  - Larsson-Sadakane algorithm : Good worst case behavior [O(NlogN)]
  - Poor cache performance

What we did

- Implemented cache-oblivious distribution sort [Frigo, Leiserson, et al] and used it in suffix sorting.
  - Found to be a factor of 3 slower than using qsort based implementation.
- Developed a cache-efficient divide and conquer suffix sorting algorithm.
  - $O(N^2 lgN)$ time and 8N extra space
- Implemented an O(n) algorithm for suffix sorting[Aluru and Ko 2003].
  - Found to be a factor of 2-3 slower than the most efficient suffix sorting algorithm available.

Incorporating cache-oblivious Distribution Sort

- Sadakane performs sorting based on 1 character comparisons
- Incorporate cache-oblivious distribution sorting of integers[1].
- Incurs $\Theta(1+(n/L)(1+\log_Z n))$cache misses

1.Matteo Frigo, Charles E. Leiserson, Harald Prokop, Sridhar Ramachandran

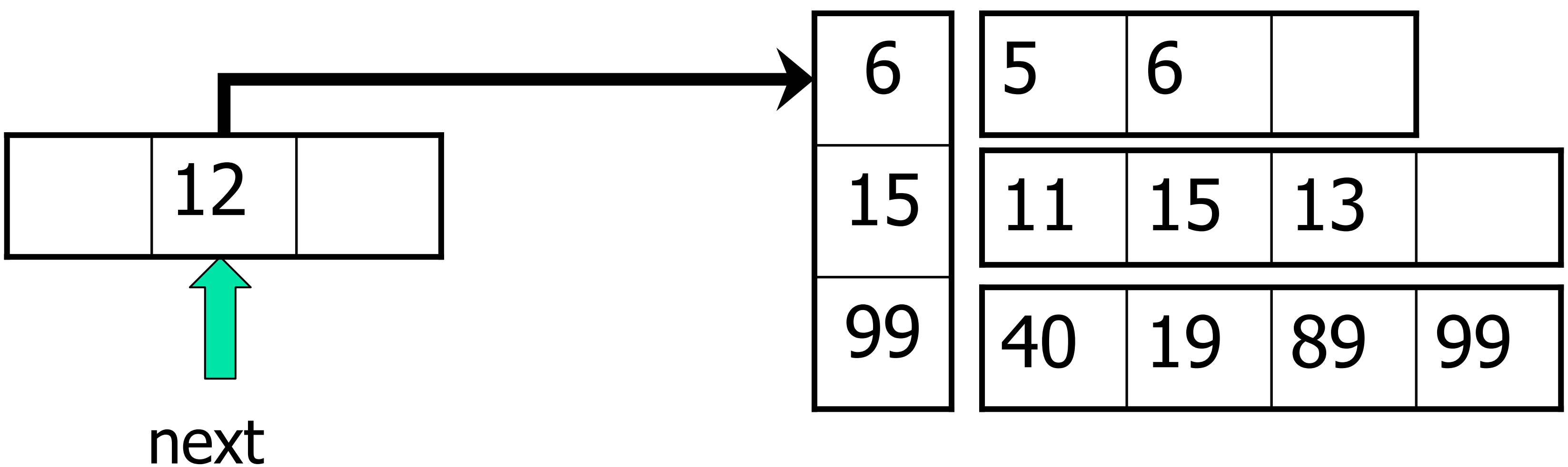
Cache-Oblivious Algorithms *FOCS 1999*

Algorithm

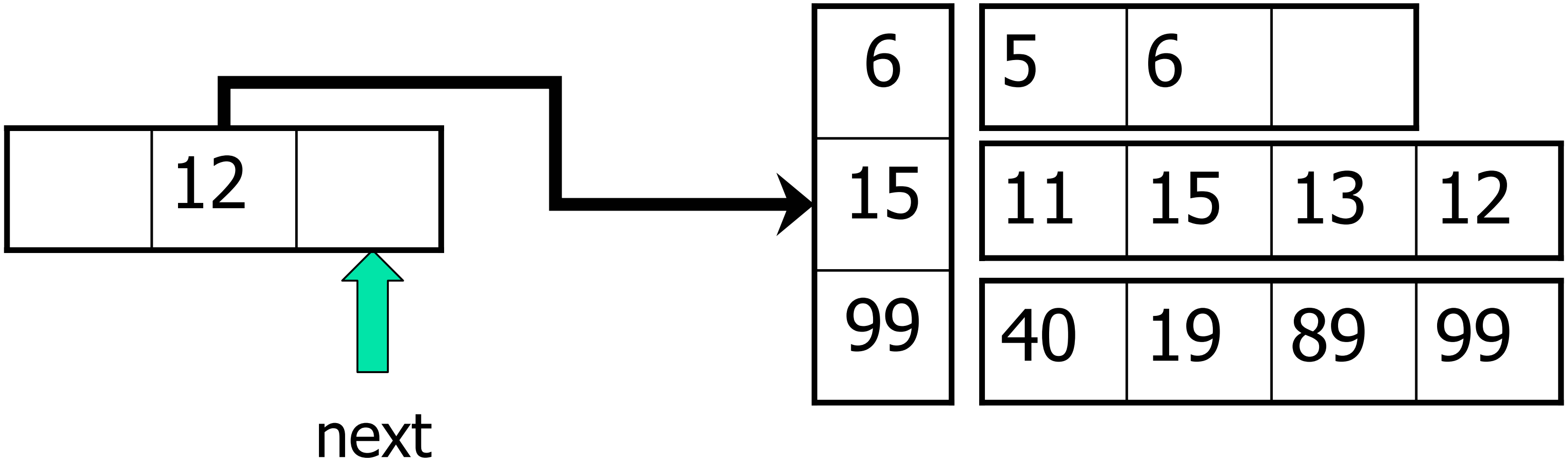
1. Partition A into √n contiguous subarrays of size √n. Recursively sort each subarray.
2. Distribute sorted subarrays into $q \leq \sqrt{n}$ buckets B_1, B_2,… B_q of size n_1, n_2,…, n_q respectively, $ for i=[1,q-1]
   a. $\max\{x|x \in B_i\} \leq \min\{x|x \in B_{i+1}\}$,
   b. $n_i \leq 2\sqrt{n}$
3. Recursively sort each bucket.
4. Copy sorted buckets back to array A.

Basic Strategy

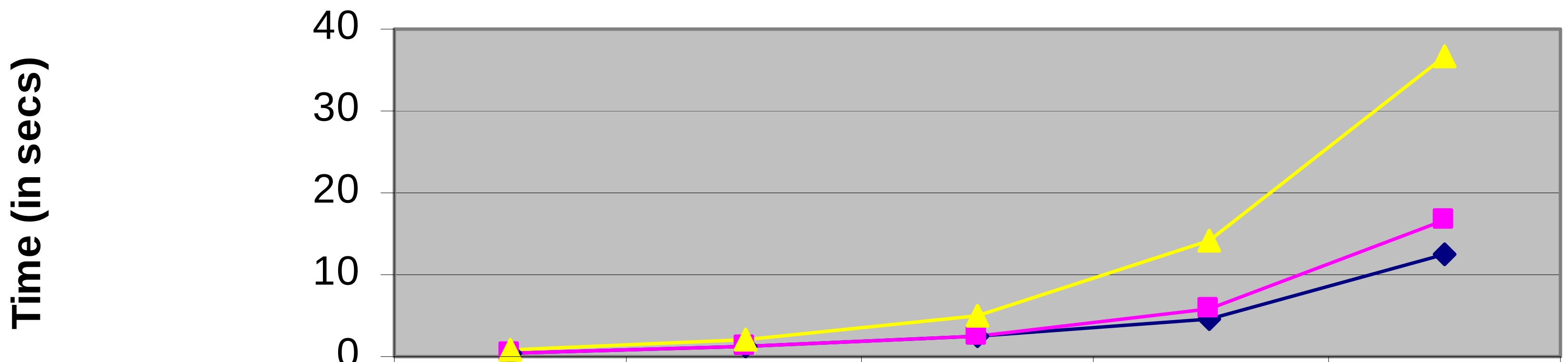
- Copy the element at position *next* of a subarray to bucket *bnum.*
- Increment *bnum* until we find a bucket for which element is smaller than *pivot* .



Bucket found...



Performance



	128KB	256KB	512KB	1MB	2MB
Qsort	0.321	1.252	2.428	4.679	12.342
MergeSort	0.491	1.171	2.623	5.808	16.833
DistriSort	0.811	1.993	4.807	14.13	36.657

Block Size

Restrictions

- mallocs caused by repeated bucket-splitting.
- Need to keep track of state information for buckets and sub-arrays
- Buckets, subarrays, copying elements back and forth incur memory management overhead.
- Splitting into $\sqrt{n} * \sqrt{n}$ subarrays, when n is not a perfect square causes rounding errors.
- Running time may not be dominated by cache misses.

Divide and conquer algorithm

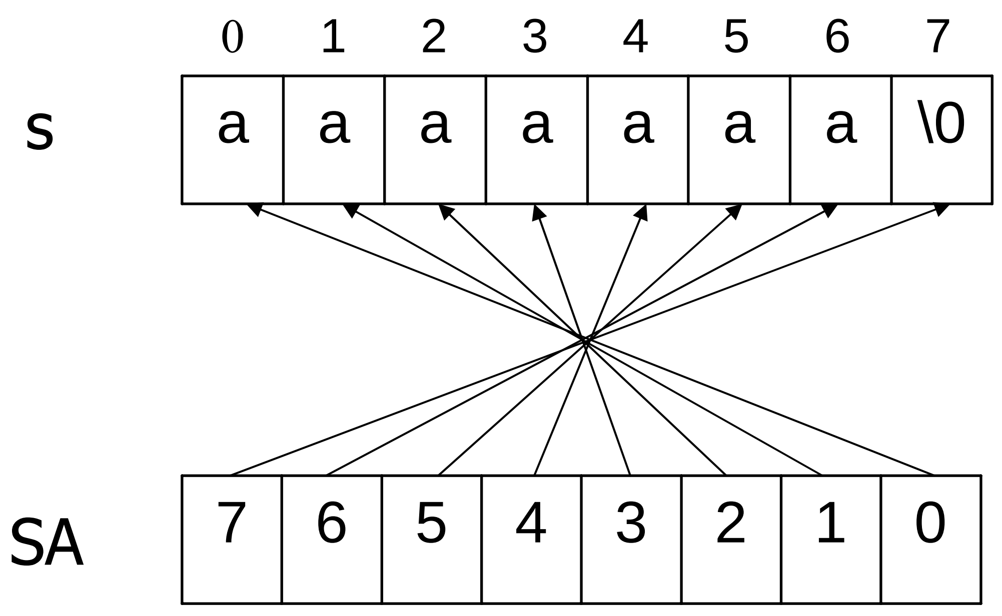
- Similar to merge sort
- Suffix sorting : From right to left
- Stores match lengths to avoid repeated comparisons in the merge phase

```
sort(char *str, int *sa, int *ml, int len){
   int mid = len/2;

   if(len <=2){
        …
   }
   …
   sort(&str[mid], &sa[mid], &ml[mid], len-mid);
   sort(str, sa, ml, mid);

   merge(s, sa, ml, len);
}
```

Suffix array



Sort phase

s

0	1	2	3	4	5	6	7
a	a	a	a	a	a	a	\0

0	1	2	3
a	a	a	\0

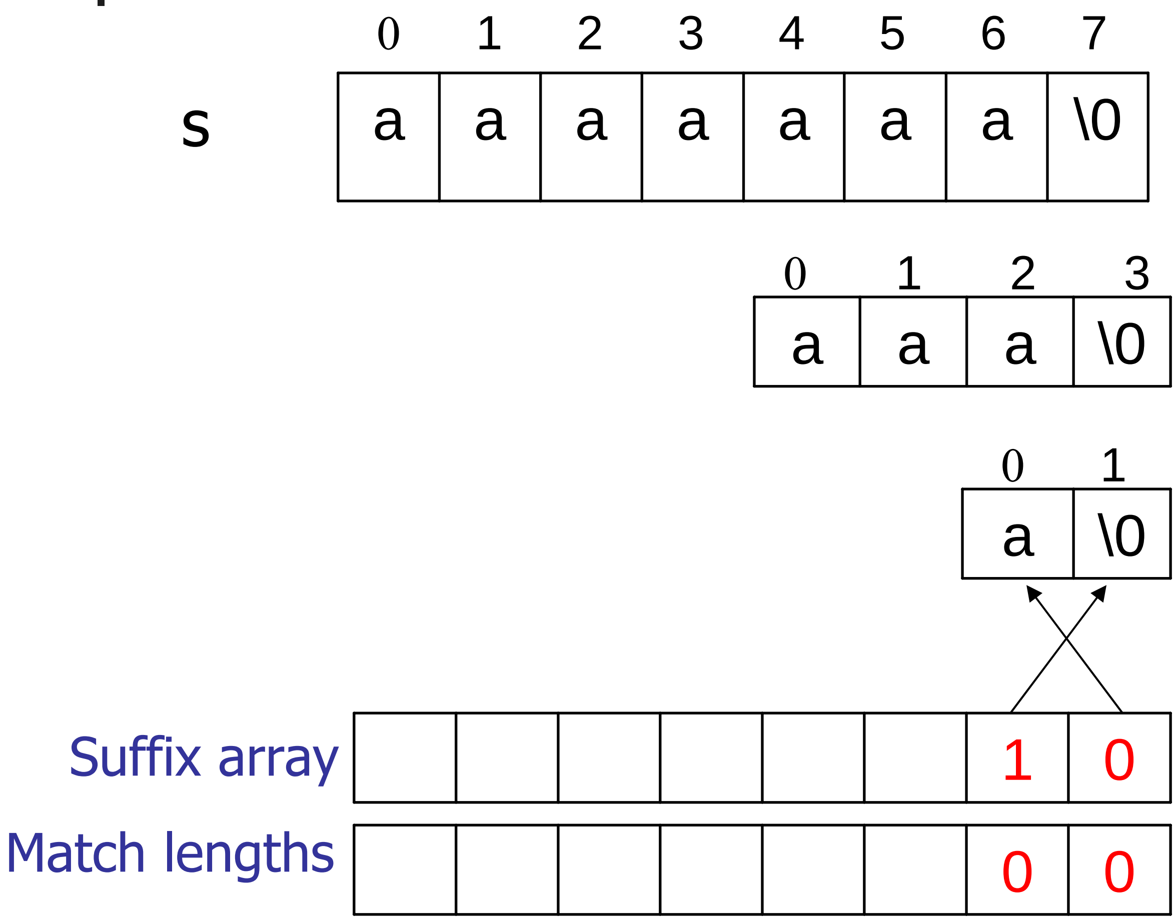
last_match_length=0

0	1
a	\0

Suffix array

Match lengths

Sort phase



last_match_length=0

Sort phase

	0	1	2	3	4	5	6	7
s	a	a	a	a	a	a	a	\0

0	1	2	3
a	a	a	\0

0	1	0	1
a	a	a	\0

last_match_length=0

Suffix array							1	0
Match lengths							0	0

Sort phase

s

0	1	2	3	4	5	6	7
a	a	a	a	a	a	a	\0

0	1	2	3
a	a	a	\0

0	1	0	1
a	a	a	\0

last_match_length=0

Suffix array							1	0
Match lengths							0	0

Sort phase

s

0	1	2	3	4	5	6	7
a	a	a	a	a	a	a	\0

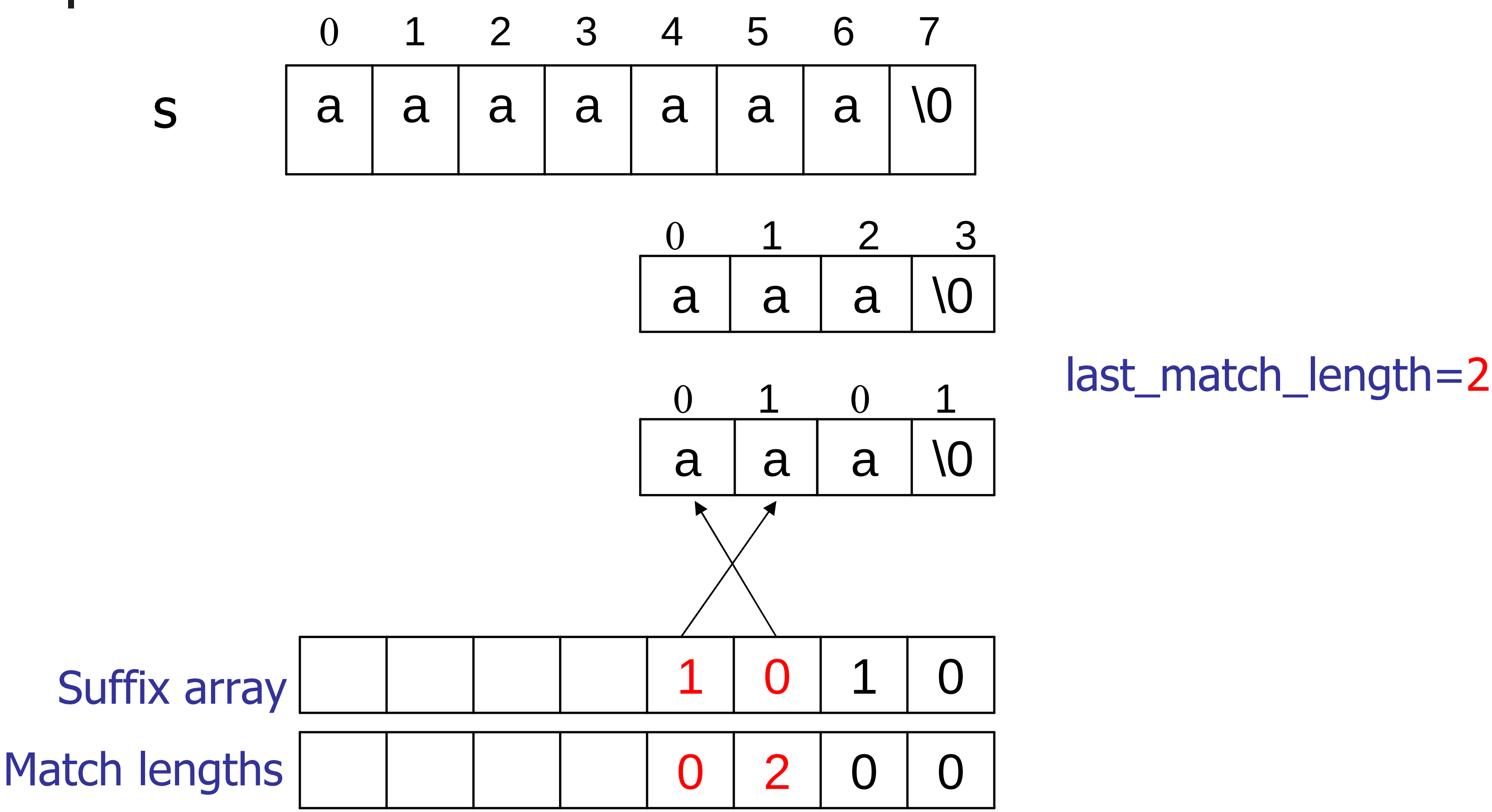
0	1	2	3
a	a	a	\0

0	1	0	1
a	a	a	\0

last_match_length=0

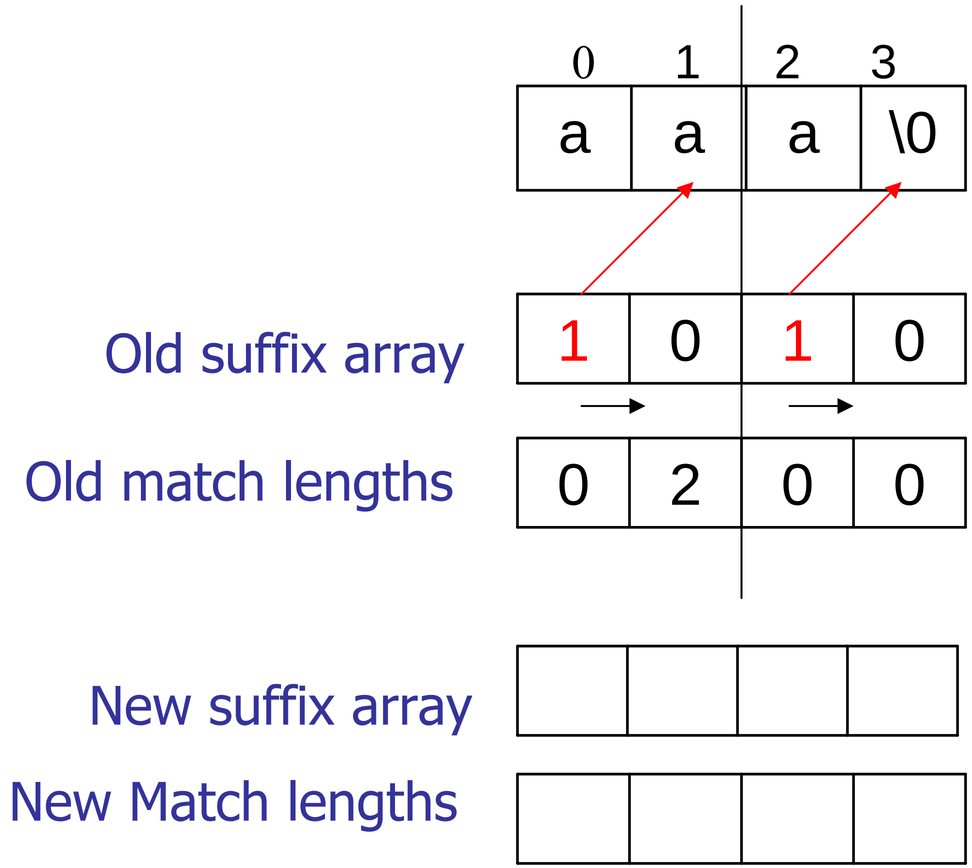
Suffix array							1	0
Match lengths							0	0

Sort phase

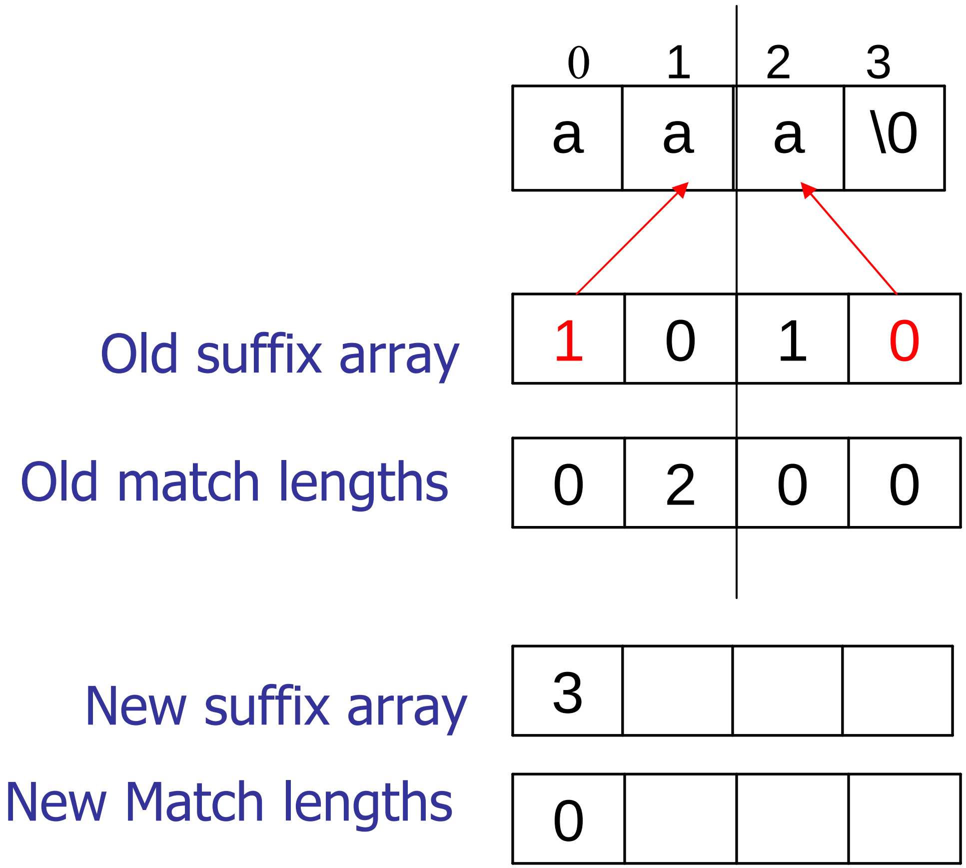


0 1 2 3 4 5 6 7
s
a a a a a a a \0
0 1 2 3
a a a \0
last_match_length=2
0 1 0 1
a a a \0
Suffix array
1 0 1 0
Match lengths
0 2 0 0

Merge phase

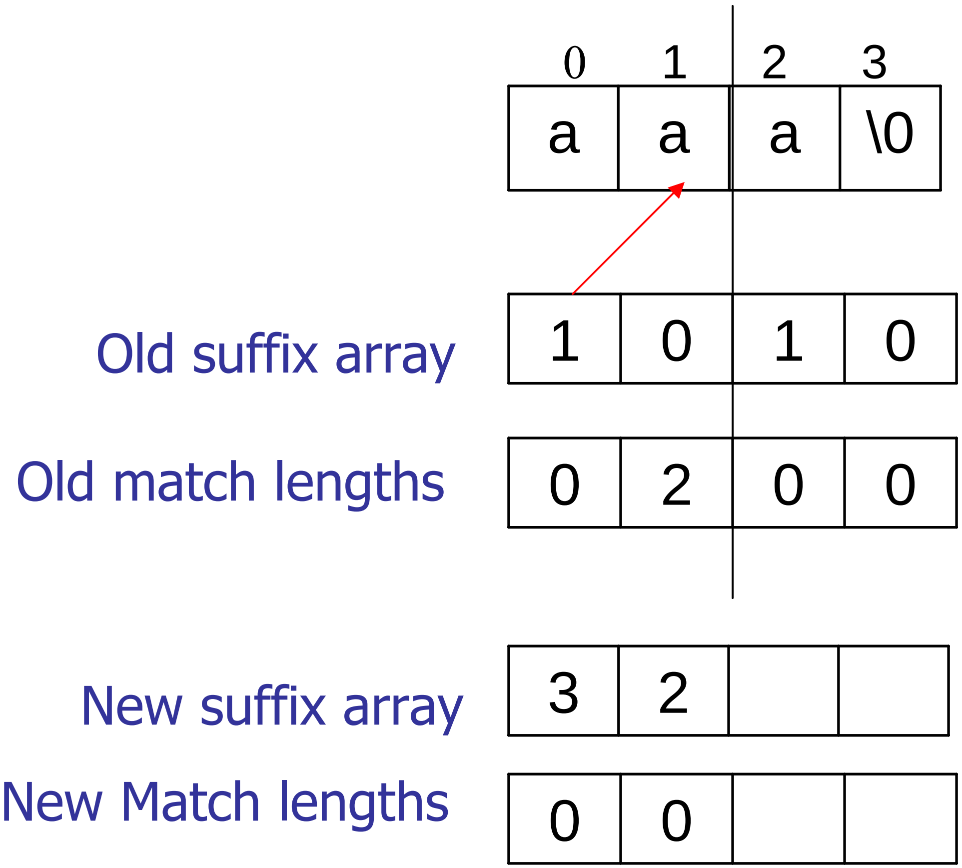

0 1 2 3
a a a \0
Old suffix array
1 0 1 0
Old match lengths
0 2 0 0
New suffix array
New Match lengths

Merge phase

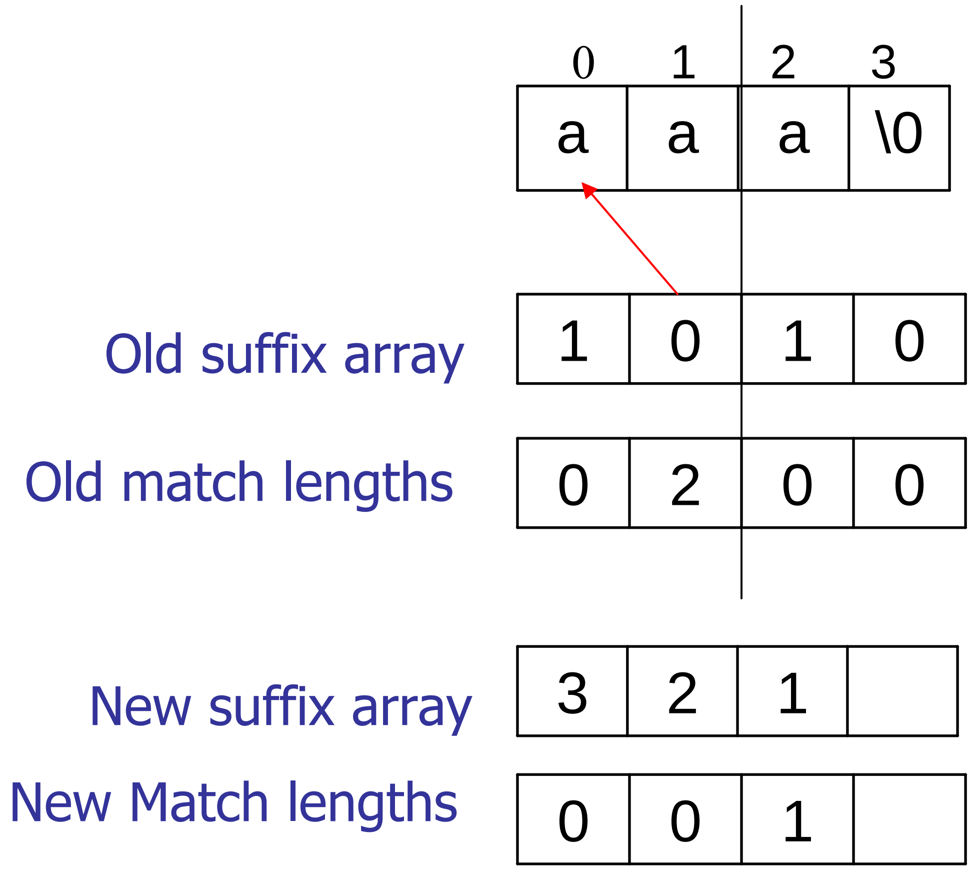


0 1 2 3
a a a \0
Old suffix array
1 0 1 0
Old match lengths
0 2 0 0
New suffix array
3
New Match lengths
0

Merge phase


0 1 2 3
a a a \0
Old suffix array
1 0 1 0
Old match lengths
0 2 0 0
New suffix array
3 2
New Match lengths
0 0

Merge phase



0 1 2 3
a a a \0
Old suffix array
1 0 1 0
Old match lengths
0 2 0 0
New suffix array
3 2 1
New Match lengths
0 0 1

Merge phase

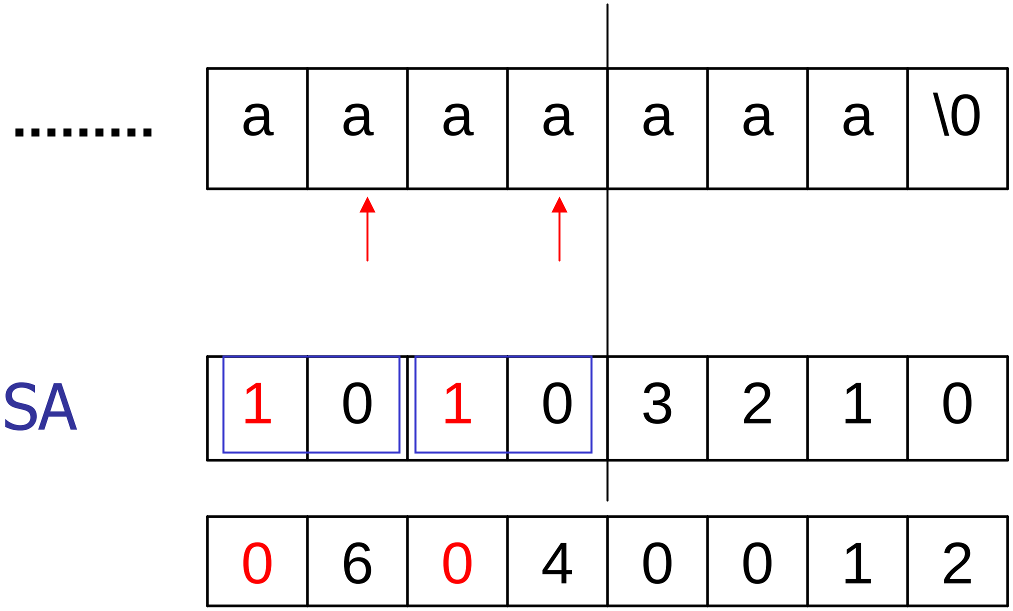
0	1	2	3
a	a	a	\0

Old suffix array	1	0	1	0
Old match lengths	0	2	0	0

New suffix array	3	2	1	0
New Match lengths	0	0	1	2

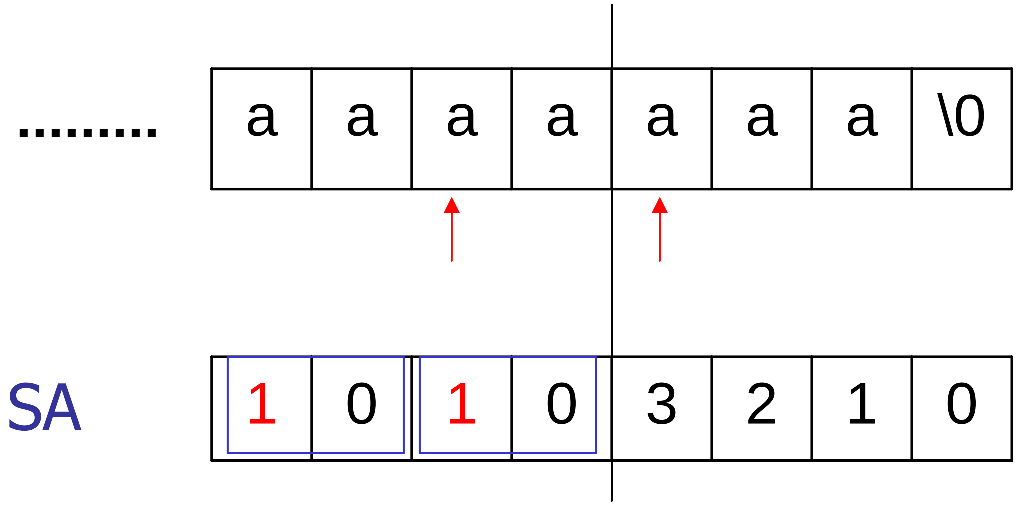
Merge phase

- Another example – large match lengths



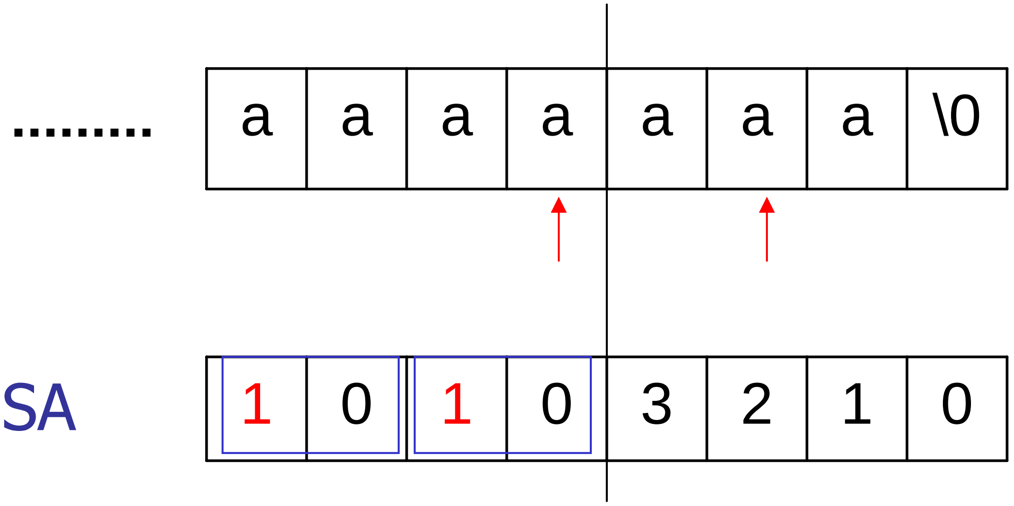
Merge phase

- Another example – large match lengths



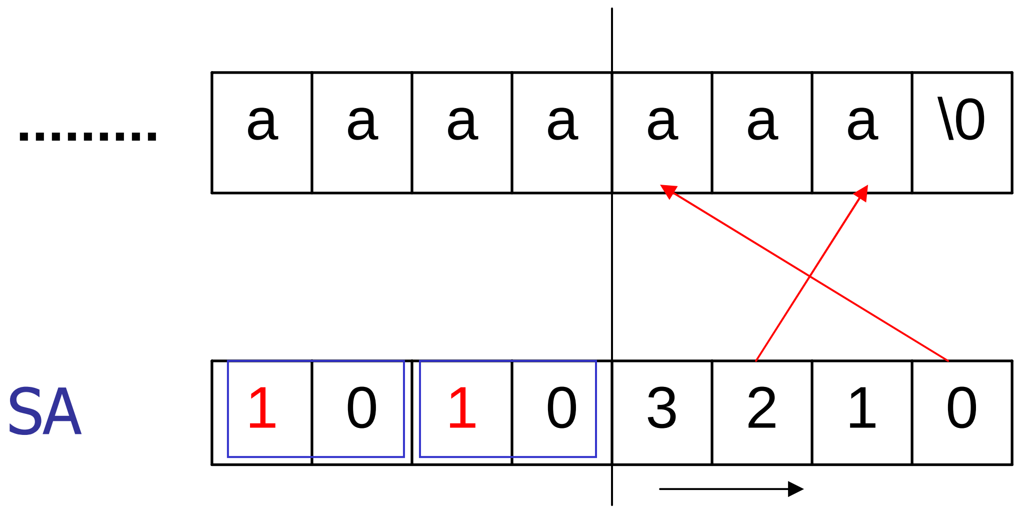
Merge phase

- Another example – large match lengths



Merge phase

- Another example – large match lengths



Analysis

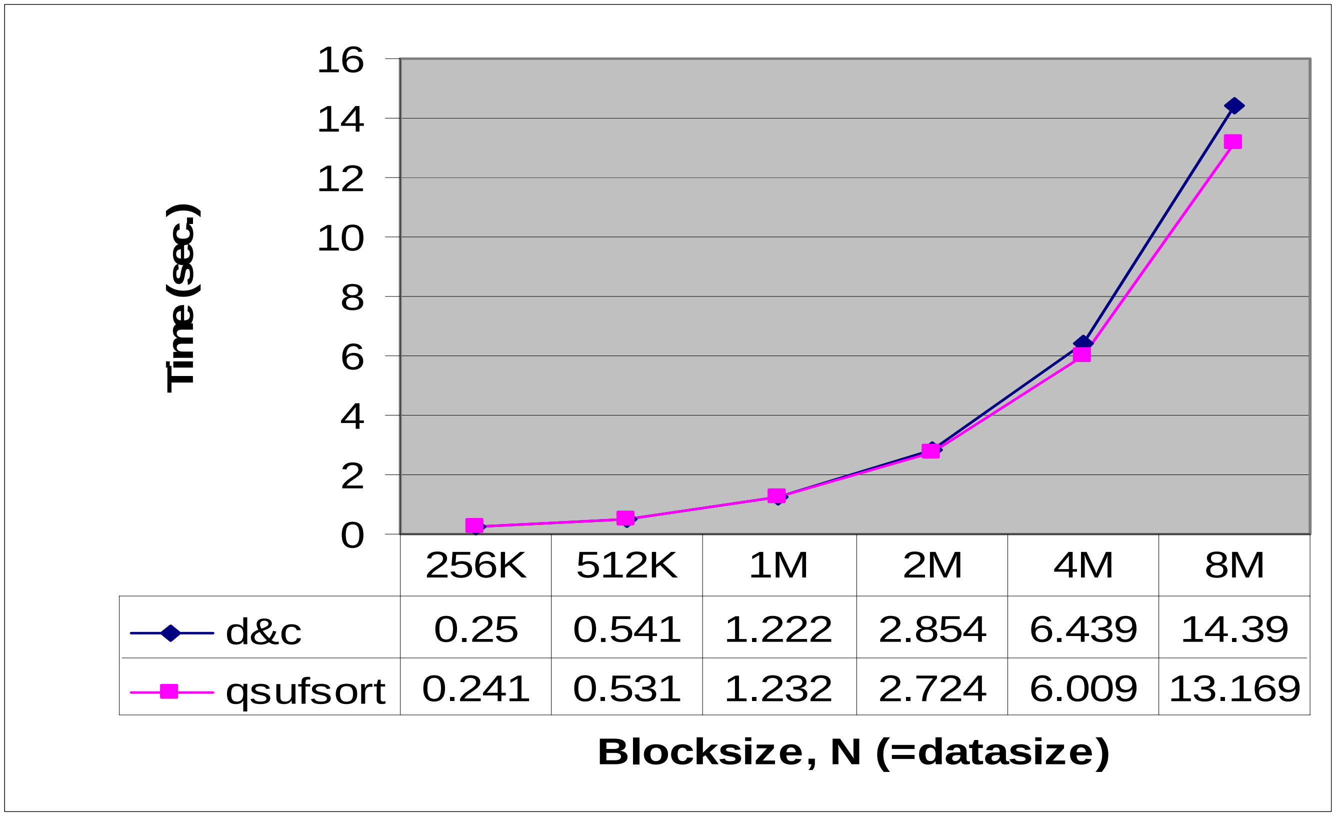
- Time complexity
  - $O(N^2 lgN)$
- Space complexity
  - 8N extra space
- O(NlgN) random I/Os

Performance

- Comparison with qsufsort[1] algorithm for suffix sorting
  - Time: O(NlogN) time
  - Space: 2 integer arrays of size N

[1]N. Larsson & K.Sadakane, Faster Suffix Sorting. Technical Report, LU_CS-TR-99-214, Dept. of Computer Science, Lund University, Sweden, 1999

Divide and conquer Vs qsufsort



	256K	512K	1M	2M	4M	8M
d&c	0.25	0.541	1.222	2.854	6.439	14.39
qsufsort	0.241	0.531	1.232	2.724	6.009	13.169

- Based on Human Genome data set
- Pentium4 2.4 GHz, 256MB RAM

Cache performance

- Blocksize N = 1,048,576
  - **Input file: reut2-013.sgm (Reuters corpus) [1MB]**

	qsufsort	d&c
# data references	525,305K	1,910,425K
L1 data cache misses	14,614K	13,707K
cache miss ratio	2.7%	0.7%

- 16KB L1 set associative data cache

Cache performance

- Blocksize N = 1,048,576
  - **Input file: nucall.seq (Protein sequence) [890KB]**

	qsufsort	d&c
# data references	531,201K	2,626,356K
L1 data cache misses	16,323K	12,945K
cache miss ratio	3.0%	0.4%

- 16KB L1 set associative data cache

Divide and conquer algorithm

- Requires no memory parameters to be set
- Good cache behavior
- Extensive testing with different types of files is needed to evaluate its utility

Linear time construction of Suffix Arrays[1]

[1]Pang Ko and Srinivas Aluru, Space Efficient Linear Time Construction of Suffix Arrays 2003

Classify as type S or L

T	M	I	S	S	I	S	S	I	P	P	I	$
Type	L	S	L	L	S	L	L	S	L	L	L	S
Pos	1	2	3	4	5	6	7	8	9	10	11	12

S: $T_i < T_{i+1}$

L: $T_{i+1} < T_i$

Sorting Type S suffixes

T	M	I	S	S	I	S	S	I	P	P	I	$
Type	L	S	L	L	S	L	L	S	L	L	L	S
Pos	1	2	3	4	5	6	7	8	9	10	11	12

Order of Type S suffixes:

12	8	2	5

Sorting Type S suffixes

T	M	I	S	S	I	S	S	I	P	P	I	$
Type	L	S	L	L	S	L	L	S	L	L	L	S
Pos	1	2	3	4	5	6	7	8	9	10	11	12
Dist	0	0	1	2	3	1	2	3	1	2	3	4

1	9	3 6
2	10	4 7
3	5 8 11	
4	12	

12	2 5 8

↓

12	8	5	2

Bucket acc. to first character

T	M	I	S	S	I	S	S	I	P	P	I	$
Type	L	S	L	L	S	L	L	S	L	L	L	S
Pos	1	2	3	4	5	6	7	8	9	10	11	12

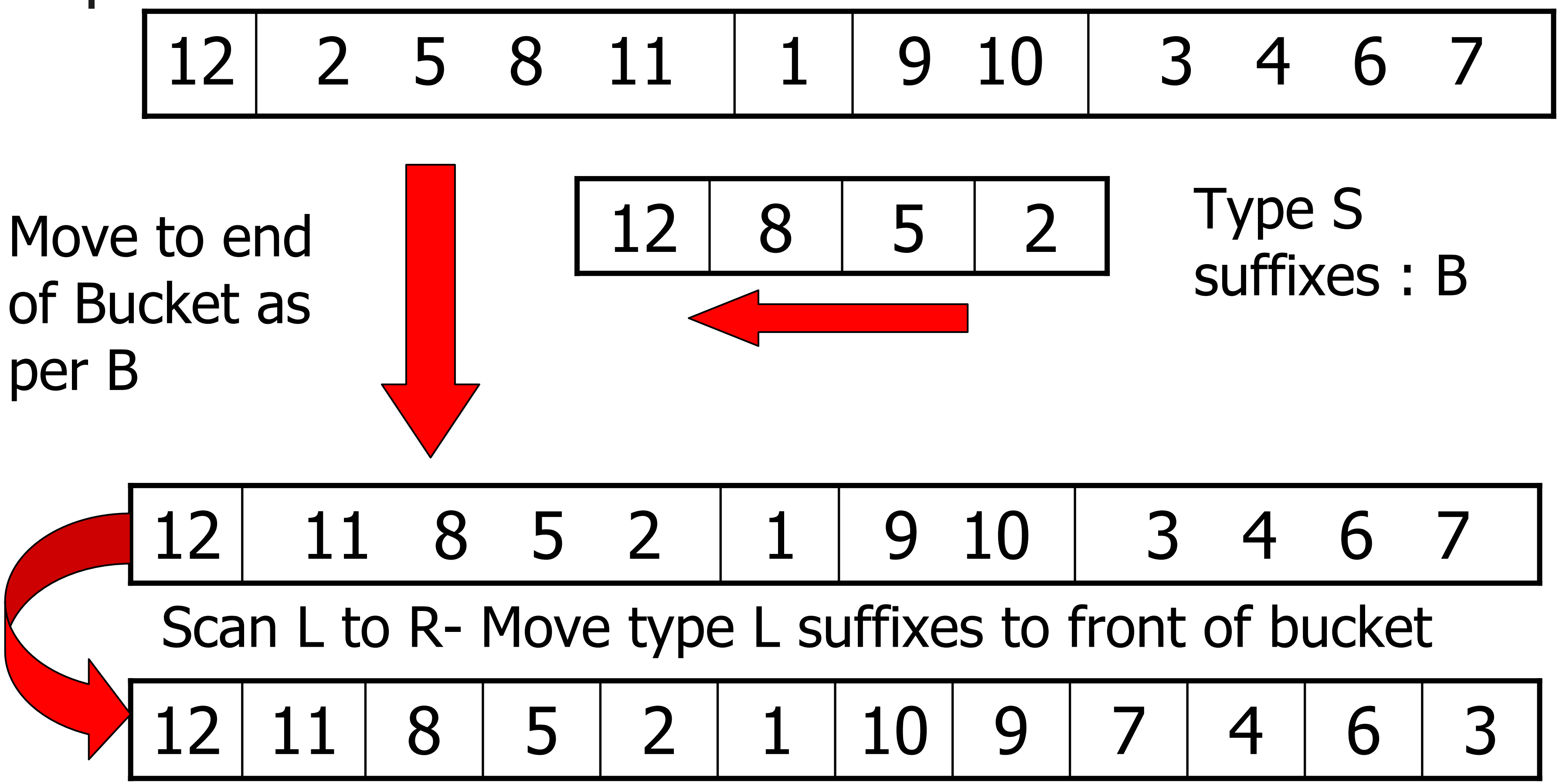
Suffix Array

$	I	M	P	S
12	2 5 8 11	1	9 10	3 4 6 7

Order of Type S suffixes

12	8	5	2

Obtaining the sorted order



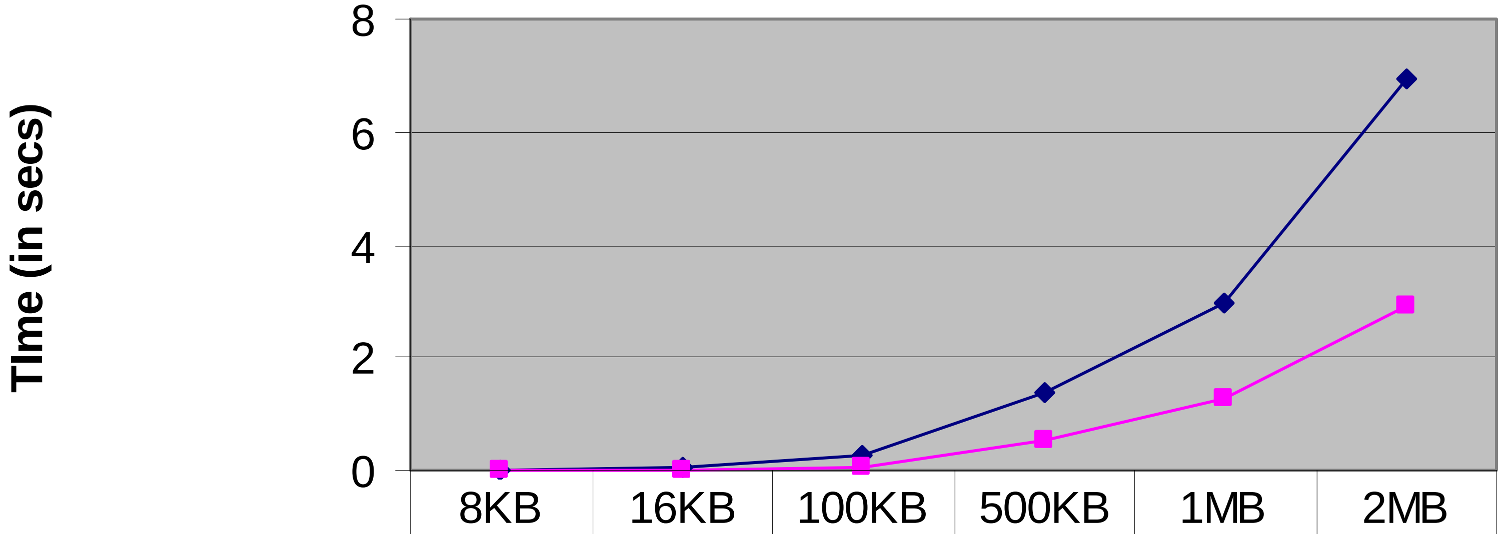
Implementation Results

Size of the file	Linear Algo	qsufsort
8KB	0.02s	0.005s
16KB	0.047s	0.01s
100KB	0.263s	0.06s
500KB	1.352s	0.551s
1MB	2.963s	1.292s
2MB	6.963s	2.894s

File used: Genome Chromosome sequence

Block Size = Size of file

Performance



	8KB	16KB	100KB	500KB	1MB	2MB
Linear Algo	0.02	0.047	0.263	1.352	2.963	6.963
qsufsort	0.005	0.01	0.06	0.551	1.292	2.894

File Size

Observations

- Using 3 integer arrays of size n, 3 boolean arrays of (2 of size n, 1 of size n/2)
- Gives rise to 12n bytes plus 2.5 bits, in comparison to 8n bytes used by Manber and Myers' O(n log n) algorithm → Trade-off between time and space.
- Implementation still crude. Further optimizations possible.
- An extra integer array to store the Reverse positions of the Suffix array in the string improves performance.

Conclusions

- The cache-oblivious Distribution Sort based suffix sorting incurs memory management overheads.
  - Factor of 3 to 4 slower than qsort based approach.
- Our Divide and conquer algorithm is cache-efficient and requires no memory parameters to be set.
  - $O(N^2 \lg N)$ time and 8N extra space
- Linear time suffix sorting algorithm's performance can be improved further. Requires more space.
  - Factor of 2 slower than qsufsort.