Crochemore's repetitions algorithm revisited - computing runs

F. Franek M. Jiang

Department of Computing and Software
McMaster University, Hamilton, Ontario

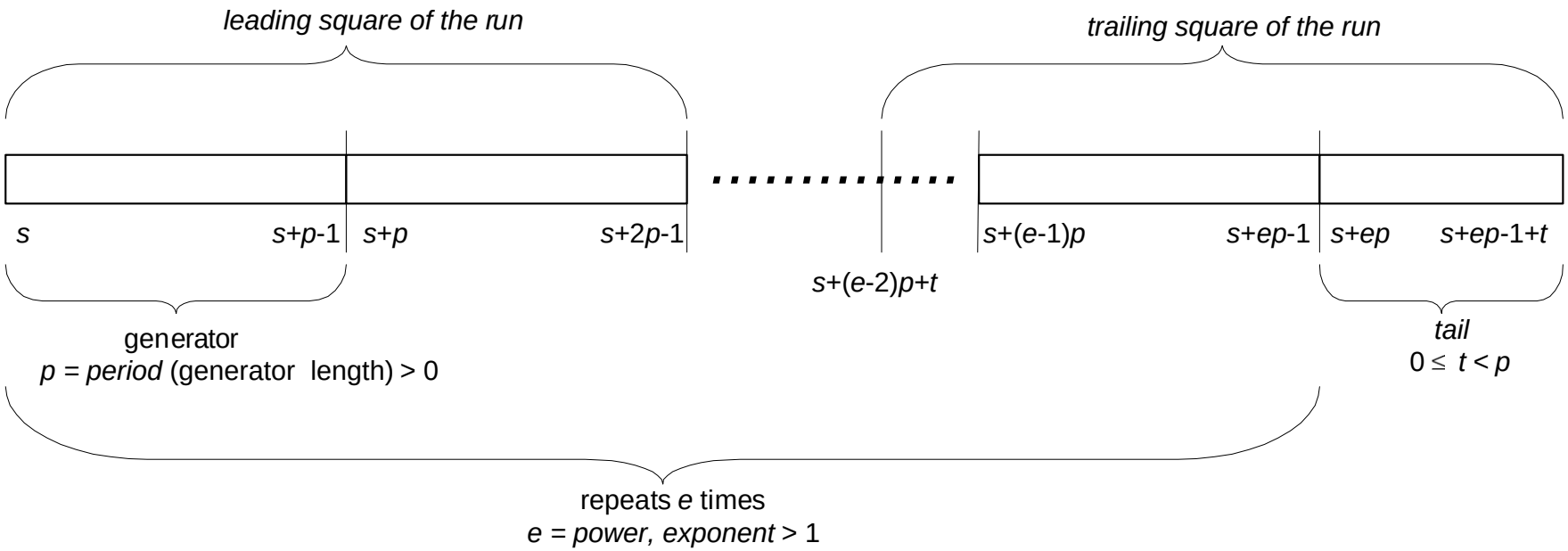
Prague Stringology Conference PSC09, Prague
August 31 - September 4, 2009

McMaster University

Outline

McMaster University

A run, a maximal fractional repetition in a string was conceptually introduced by *Main* in 1989. The term *run* was coined by *Iliopoulos*, *Moore*, and *Smyth* in 1997.



Computing runs in linear time

Main gave the following "blueprint" for an algorithm to compute the leftmost occurrence of every run in a string x:

1. Compute a suffix tree of x
2. using the suffix tree, compute Lempel-Ziv factorization of x (linear, *Lempel* and *Ziv*)
3. using the Lempel-Ziv factorization, compute the leftmost runs (linear, *Main*)

Farach 1997 gave a linear-time algorithm to compute suffix tree. *Kolpakov* and *Kucherov* 2000 showed how to compute all runs in x from the leftmost ones in linear time.

McMaster University

Suffix trees and Farach's algorithm are not very practical for this task, but suffix arrays are:

- Lempel-Ziv factorization can be computed in linear time using suffix array (*Abouelhoda at al.* 2004)
- Suffix array can be computed in linear time (*Kärkkäinen+Sanders* 2003, *Ko+Aluru* 2003)

Thus, a practical blueprint for a linear time algorithm for computing runs:

1. Compute a suffix array of x
2. using the suffix array, compute the Lempel-Ziv factorization of x
3. using the Lempel-Ziv factorization, compute the leftmost runs
4. using the leftmost runs, compute all runs

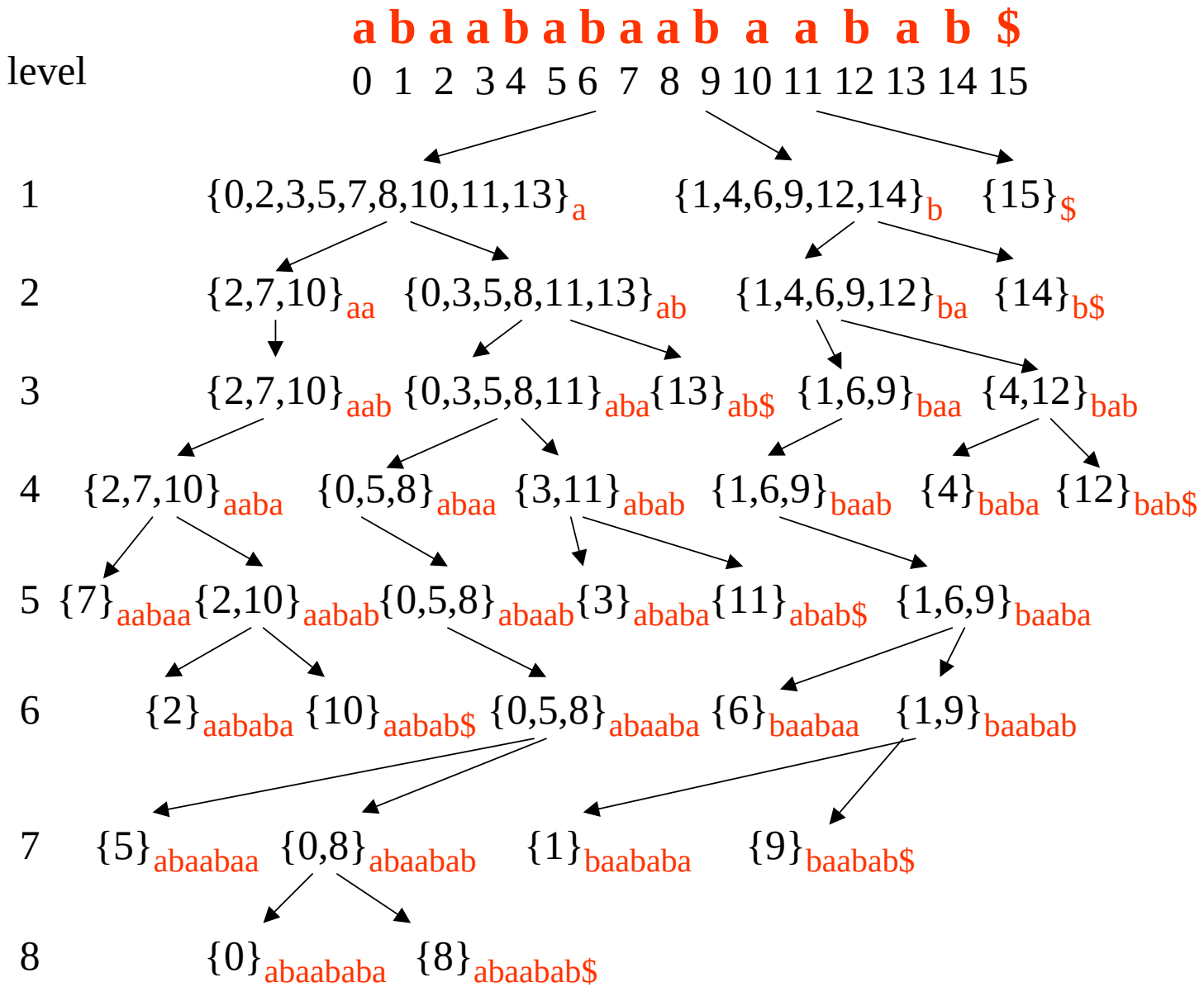
Current implementations along these lines – Johannes Fischer's at Universität Tübingen, Kucherov's at CNRS Lille, and CPS by G. Chen, S.J. Puglisi & W.F. Smyth

McMaster University

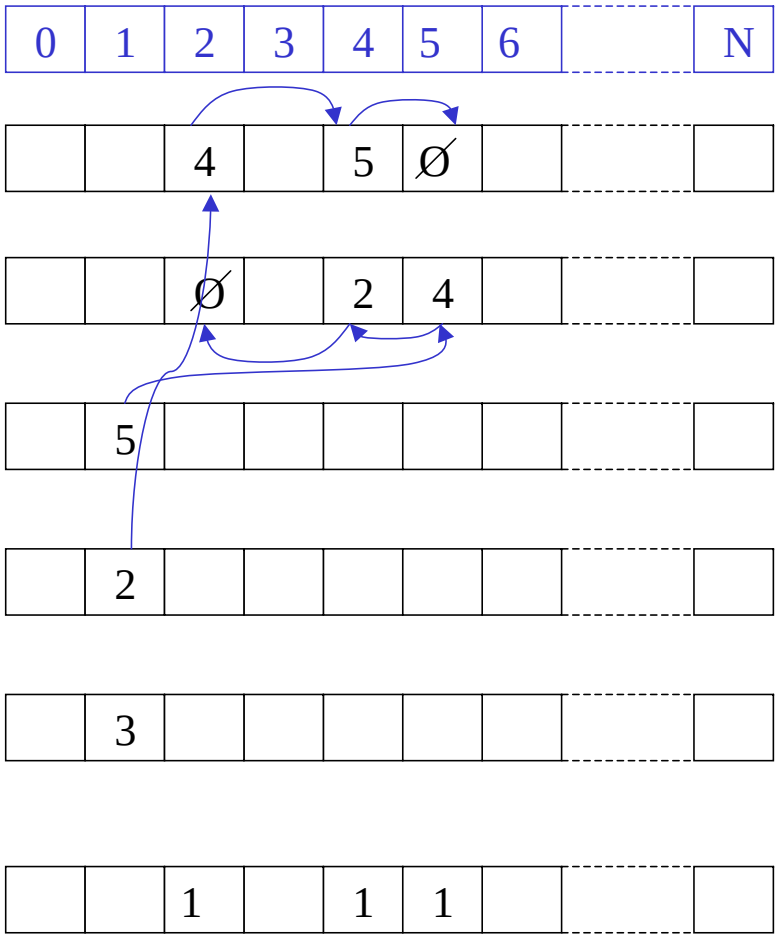
Why we are interested in Crochemore's repetitions algorithm, thought it has "only" $O(n \log n)$ complexity.

- The linear time strategy just discussed is complicated and elaborate and does not lend itself well to parallelization due to the recursive nature of the (linear) computation of the suffix array.
- The "heart" of Crochemore's repetitions algorithm, the refinement of the classes can be done naturally in parallel as the refinement of one class is independent from the refinement of another class.
- We have a good and "space efficient" implementation of Crochemore's algorithm.
- A modified Crochemore's algorithm may outperform the linear implementations (at least on some classes of strings).

McMaster University

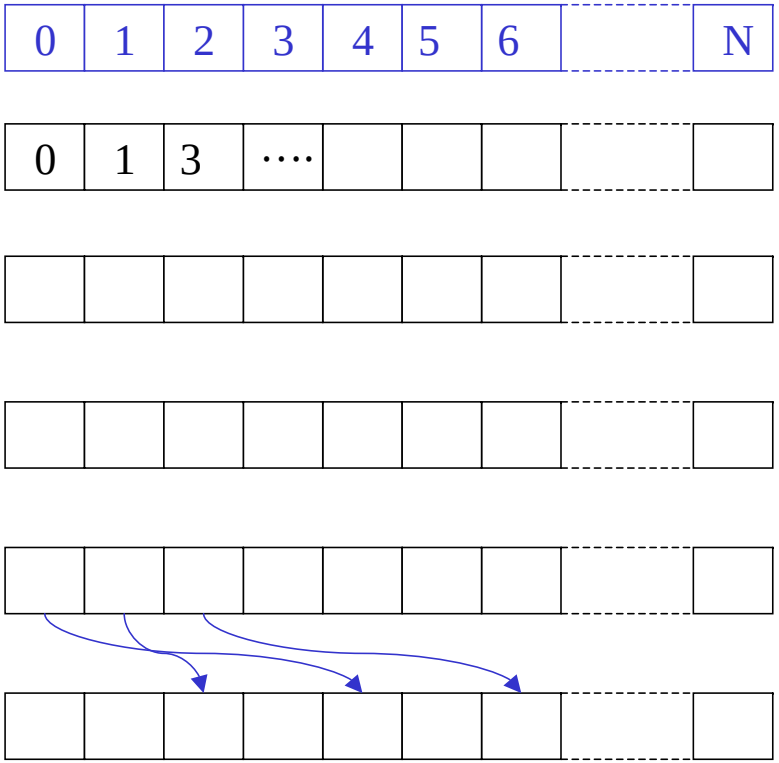

a b a a b a b a a b a a b a b $
level
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 {0,2,3,5,7,8,10,11,13}a {1,4,6,9,12,14}b {15}$
2 {2,7,10}aa {0,3,5,8,11,13}ab {1,4,6,9,12}ba {14}b$
3 {2,7,10}aab {0,3,5,8,11}aba {13}ab$ {1,6,9}baa {4,12}bab
4 {2,7,10}aaba {0,5,8}abaa {3,11}abab {1,6,9}baab {4}baba {12}bab$
5 {7}aabaa {2,10}aabab {0,5,8}abaab {3}ababa {11}abab$ {1,6,9}baaba
6 {2}aababa {10}aabab$ {0,5,8}abaaba {6}baabaa {1,9}baabab
7 {5}abaabaa {0,8}abaabab {1}baababa {9}baabab$
8 {0}abaababa {8}abaabab$

Our implementation

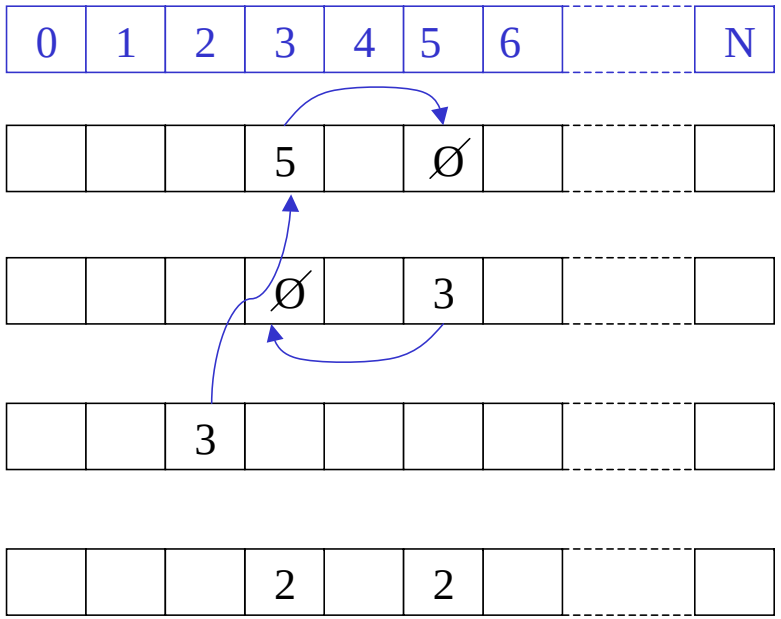


$c_1=\{2,4,5\}$

Total this slide 6*N
subtotal 6*N

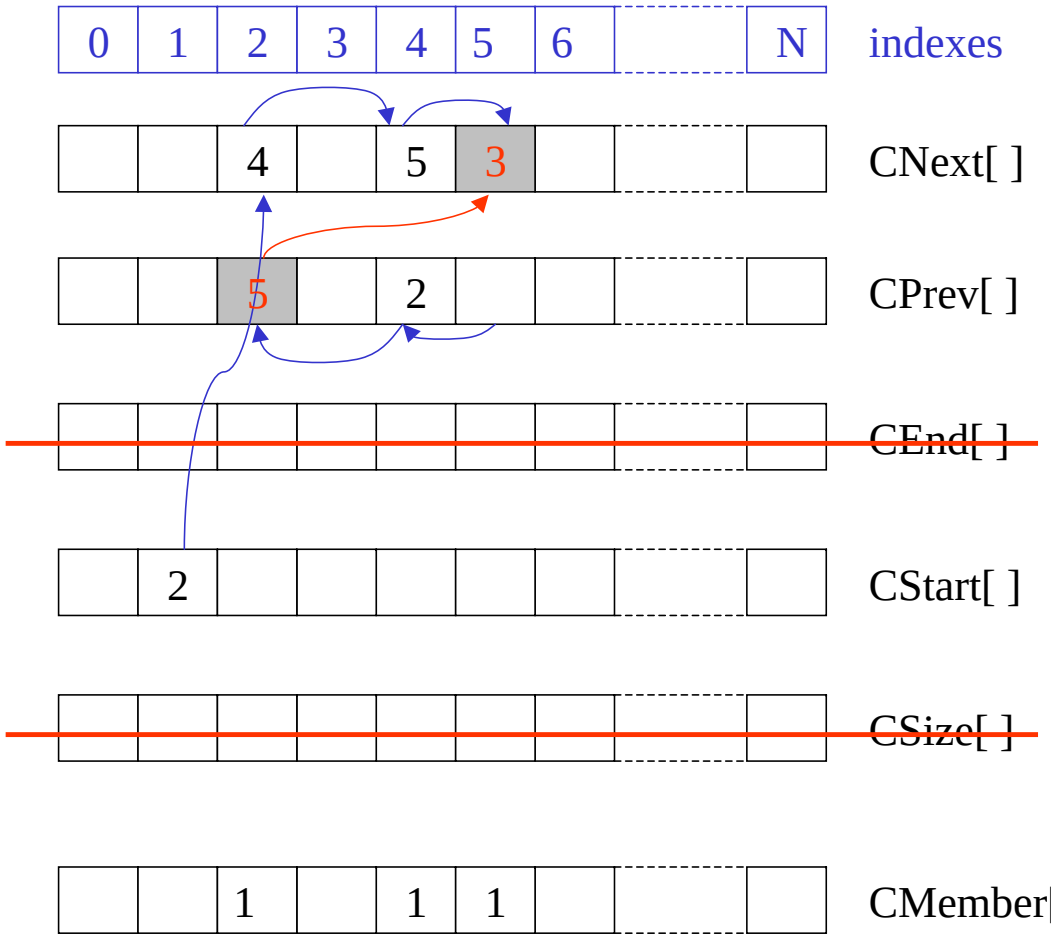


Total this slide 5*N
subtotal 11*N



$f_2=\{3,5\}$

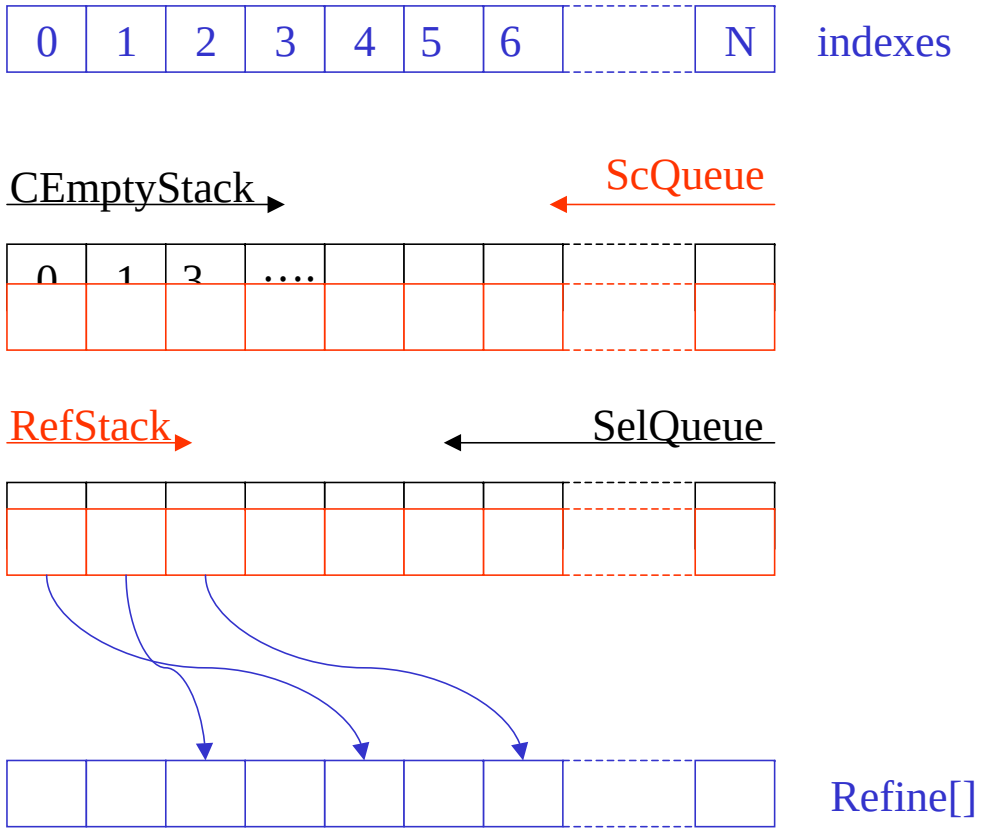
Total this slide 4*N
overall total 15*N



$c_1=\{2,4,5\}$

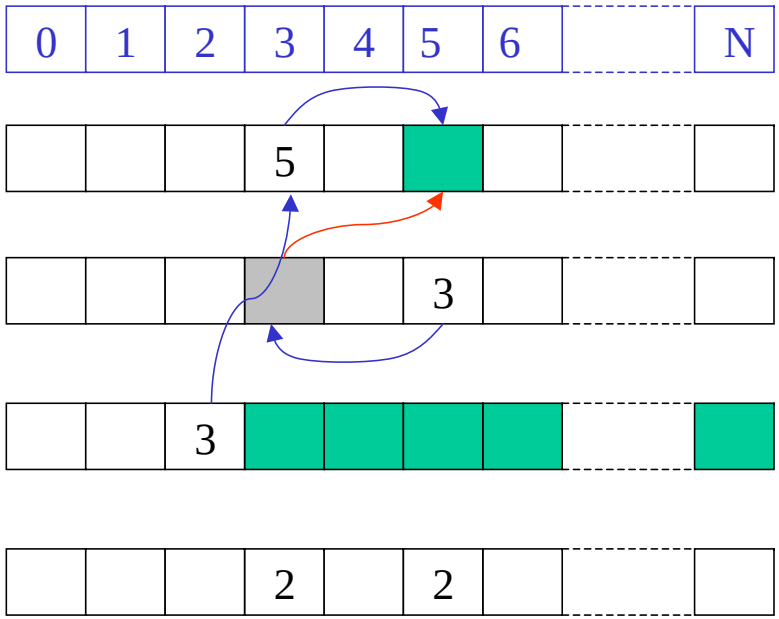
Memory virtualization

Total this slide 4*N
subtotal 4*N



Refine[] is virtualized over FNext[], FPrev[], and FStart[]

Total this slide 2*N
subtotal 6*N

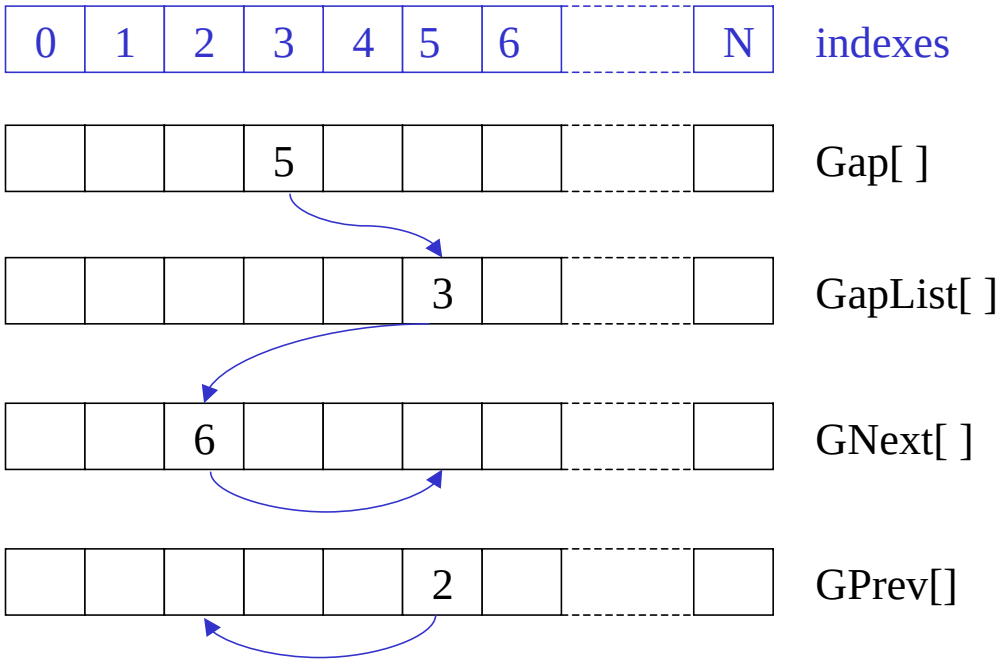


$f_2=\{3,5\}$

Memory virtualization

Refine[] is virtualized over

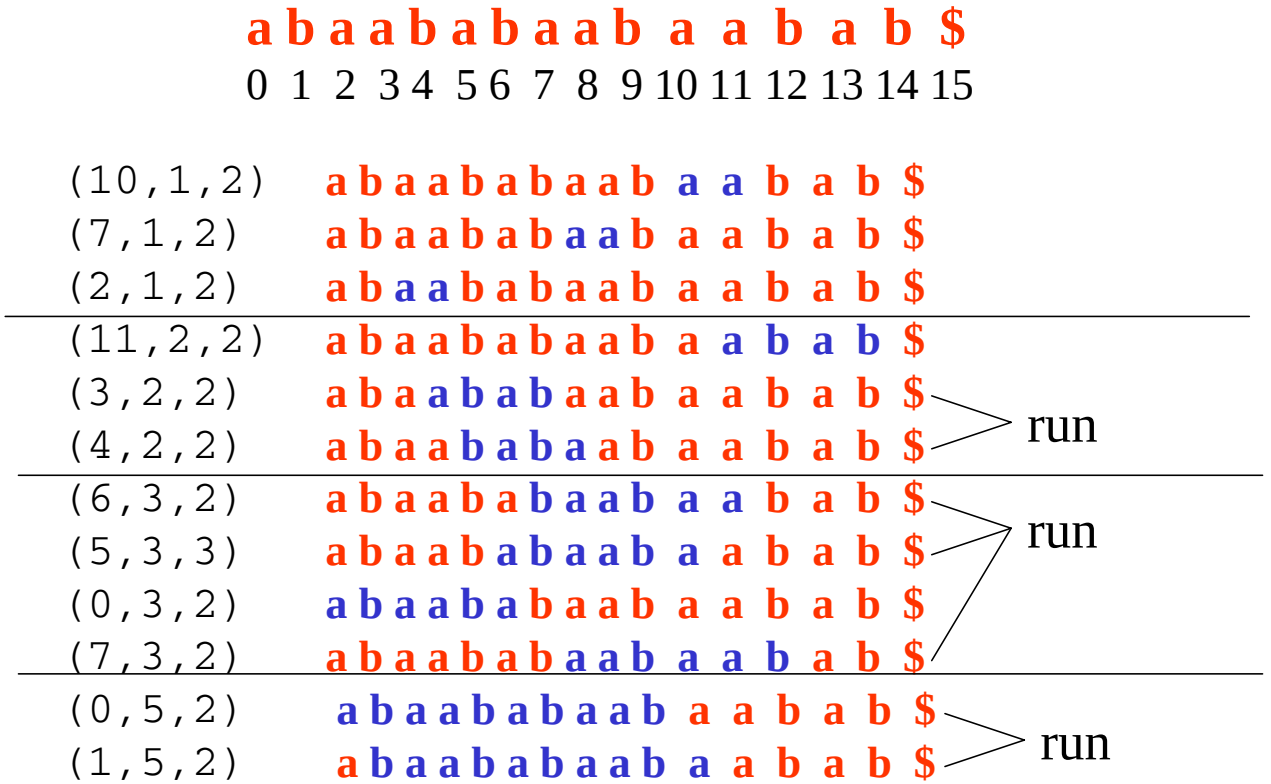
Total this slide 4*N
overall total 10*N



Total this slide 4*N
overall total 14*N

What needs to be modified

The algorithm outputs the repetitions grouped by the period (good), but regardless of the positions of the repetitions (bad)



Thus, the repetitions must be “collected" and “consolidated" into runs.

The following three variants differ in the process of “collection" and the process of “consolidation".

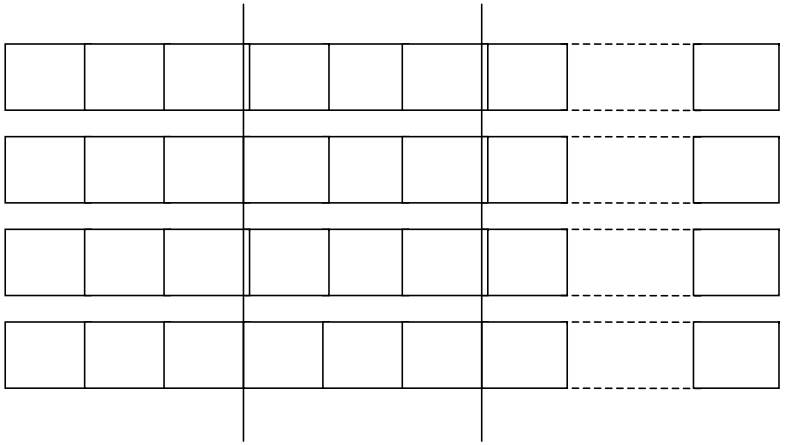
Simple modifications that worsen the complexity - Variant A

We collect the repetitions of the same period (during the processing of one level) and consolidate them immediately into runs stored in a search tree according to the starting position. The runs are reported when a level is processed and the search tree is not preserved (the data structure is reused for the next level).

When descending the tree, we compare the current repetition with the run stored in the node. If the repetition extends the run, the run is updated and the search is terminated.

If we reach the "destination", the repetition is stored in a new node as a run with 0 tail.

Data structure for variant A:



RunLeft[ ] (reuse FNext[ ])

RunRight[ ] (reuse FPrev[ ])

Run_s[ ] (reuse FMember[ ])

Run_end[ ] (reuse FStart[ ])

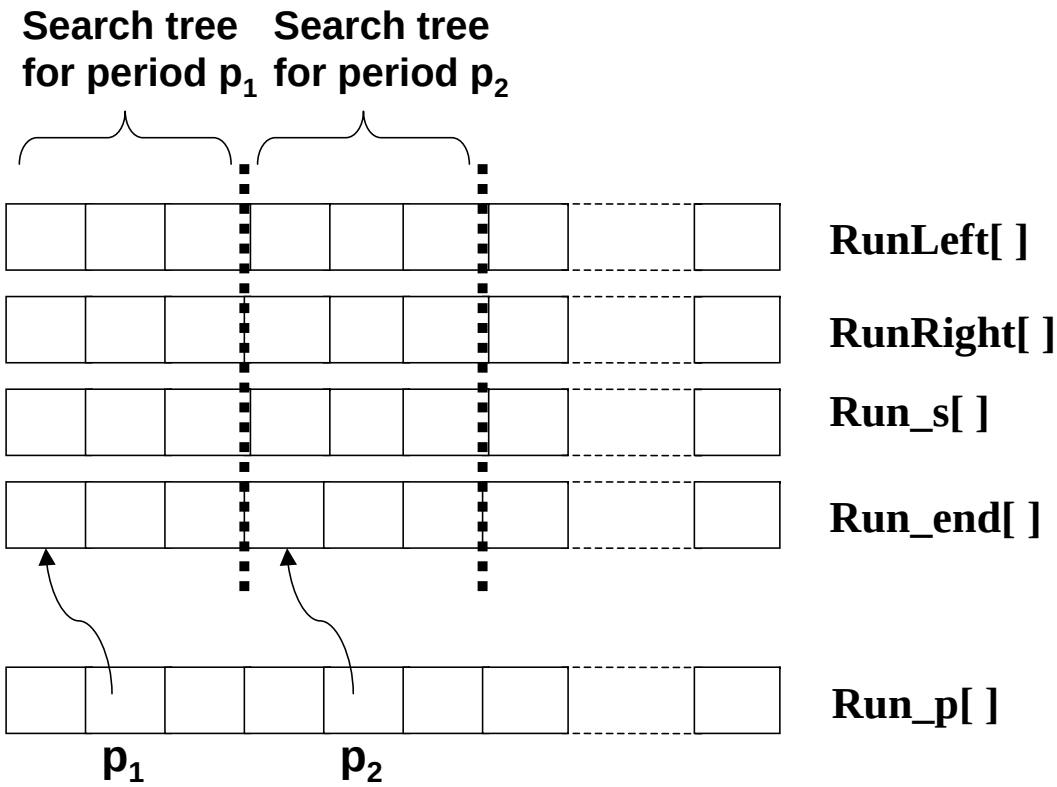
To save space, we store the runs as (*start*,*end*). The period does not need to be stored, as all the runs are of the same period.
No extra memory required, however the complexity is increased to $O(n(\log n)^2)$.

McMaster University

Simple modifications that worsen the complexity - Variant B

We collect the repetitions and consolidate them immediately into runs stored in search trees according to the starting position. All runs of the same period are collected in the same tree. The runs are reported when all levels are processed, and hence all search trees must be preserved during the processing. The rules of “consolidation" are the same as in variant A.

Data structure for variant B:



$5n$ integers of extra memory required, the complexity is increased to $O(n(\log n)^2)$.

Are we guaranteed that $5n$ integers is enough memory for all the search trees?

No, but

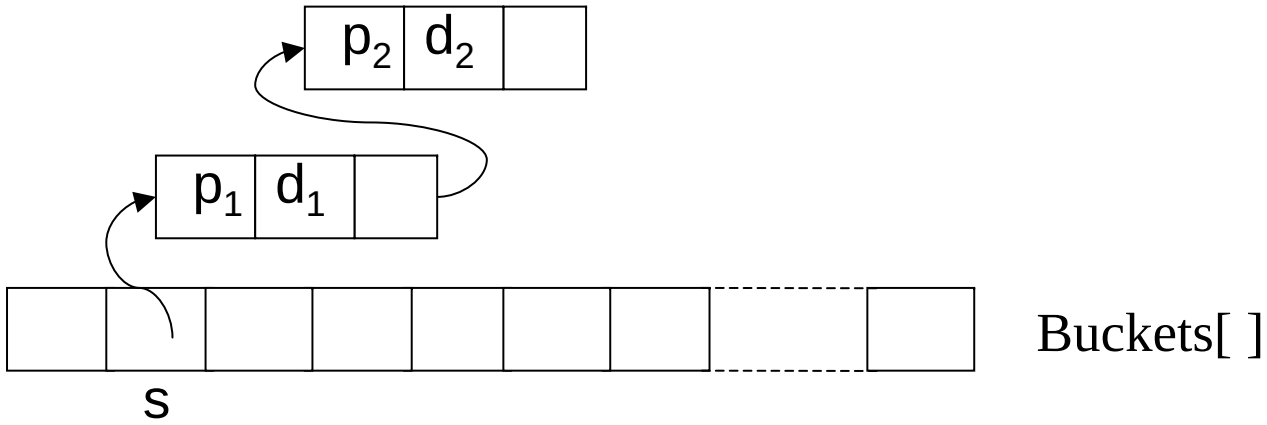
(a) we believe in the Maximum Run Conjecture that stipulates that there are at most n runs, in which case the space would be sufficient,

(b) and if our program crashes at this point, we have found a counter-example to the Maximum Run Conjecture, a nice consolation price for the crash.

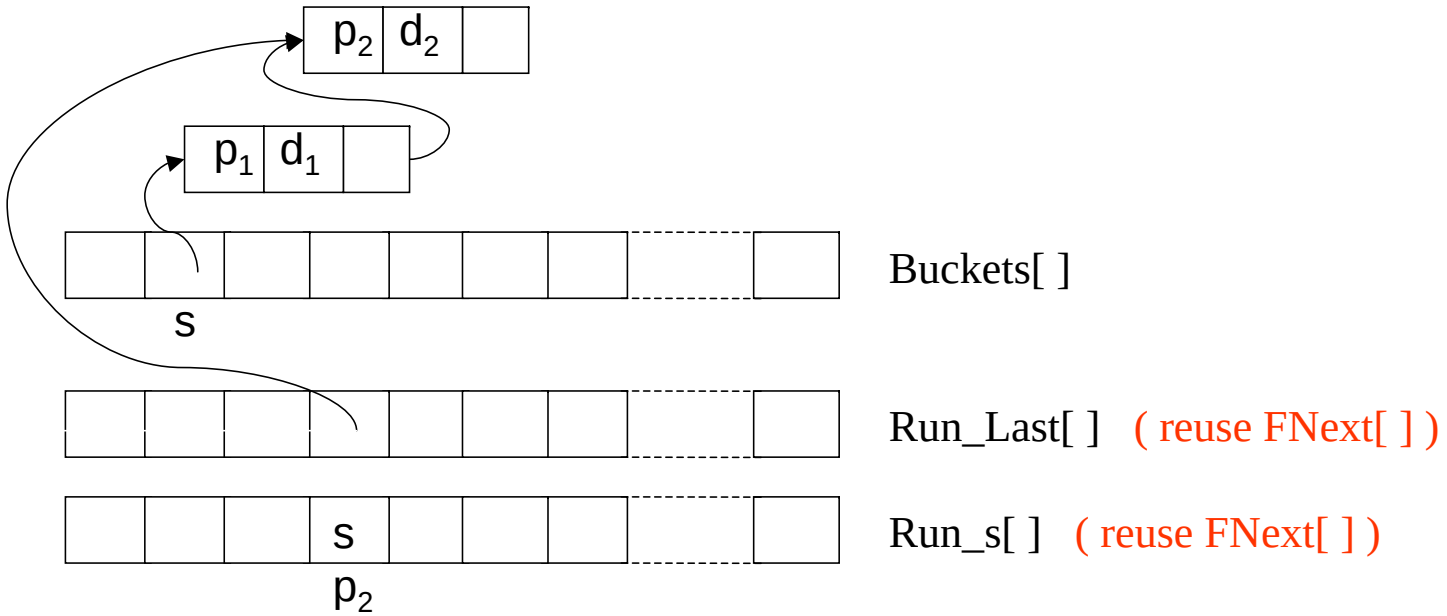
A modification that preserves the complexity - Variant C

We collect all repetitions according their starting positions in n buckets and do not consolidate them at all during the processing of the levels.

This requires at most $n \log n$ integers of additional memory.



When all levels have been processed and all repetitions collected in the buckets, we sweep the buckets from left to right and consolidate the repetitions into runs.



Test Results

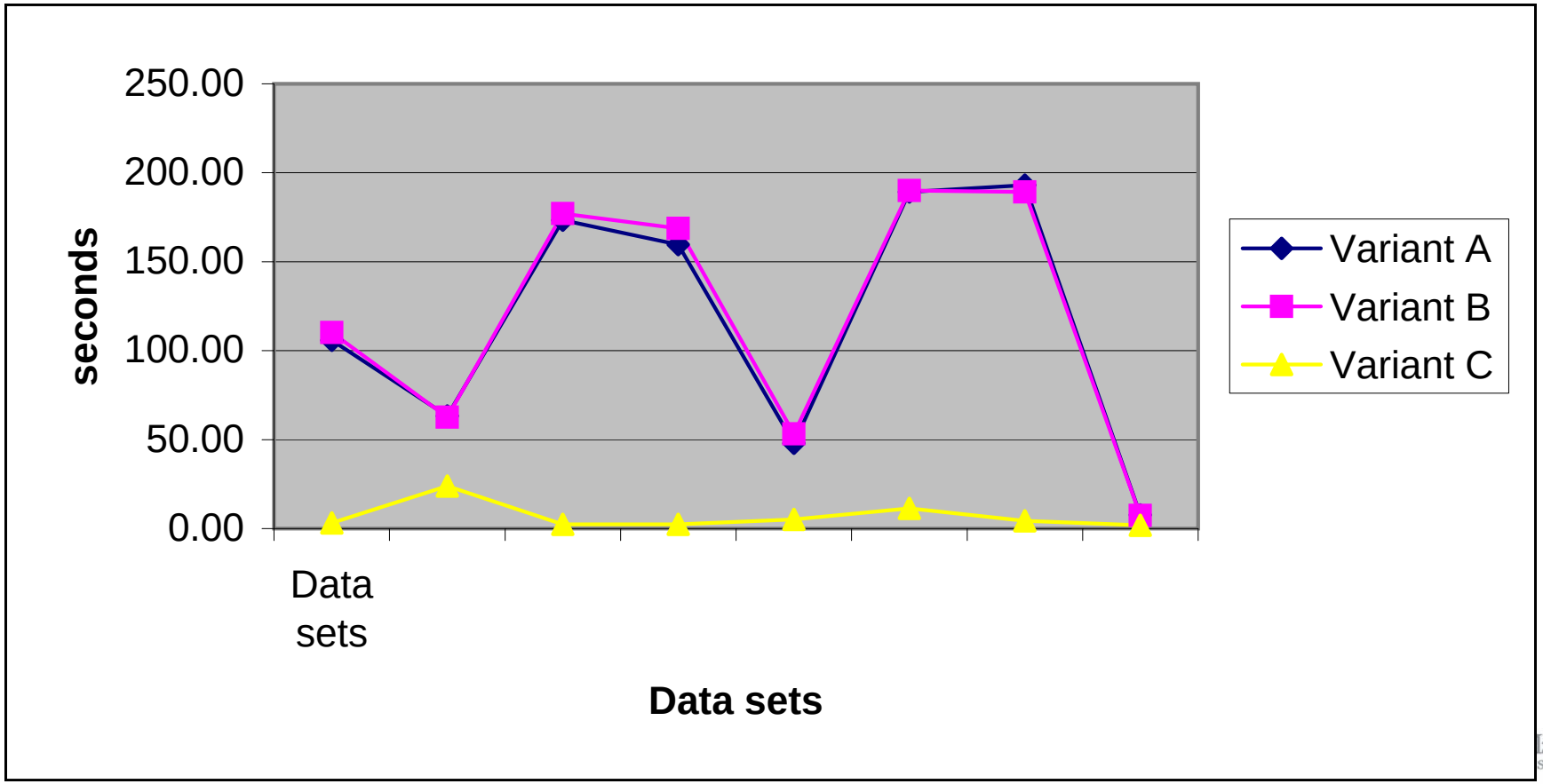
- Implementations of the three variants were compared as to their performance. The testing was rather informal, just to give indications how the three variants compare.
- Hardware: Sony VAIO laptop with Intel Core-2 Duo CPU T5800 @ 2.00GHz, 4GB of RAM
- Software: Windows Vista Home Premium SP1. The code was written in C++ and was compiled using the GNU g++ compiler.
- Each run was repeated five times, the minimum numbers were recorded.

Comparing speed performance

	Data Set	File Name	String Length	Time (seconds)		
				Variant A	Variant B	Variant C
1	DNA	dna.dna4	510976	105.87	110.15	3.12
2	English	bible.txt	4047392	63.27	62.65	23.90
3	Fibonacci	fibo.txt	305260	173.30	177.00	2.39
4	Periodic	fss.txt	304118	159.61	168.78	2.44
5	Protein	p1Mb.txt	1048576	47.93	53.23	5.15
6	Protein	p2Mb.txt	2097152	189.20	189.98	11.42
7	Random	random2.txt	510703	193.01	189.28	4.42
8	Random	random21.txt	510703	7.69	7.46	1.89

McMaster University

Comparing speed performance

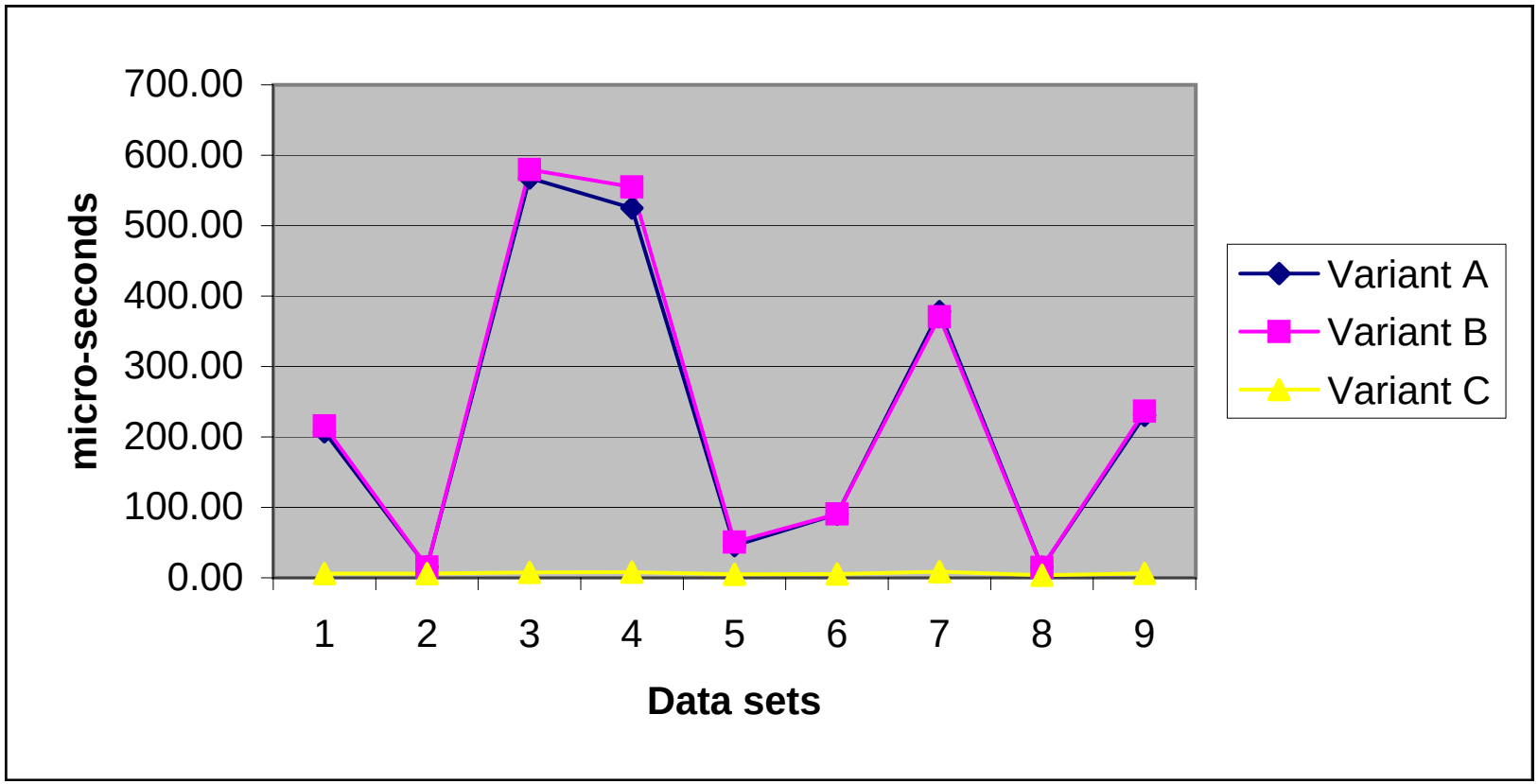


Comparing speed performance per character of input

	Data Set	File Name	Name	# of runs	Time (µsec / letter)		
					Variant A	Variant B	Variant C
1	DNA	dna.dna4	510976	130368	207.18	215.57	6.11
2	English	bible.txt	4047392	63690	15.63	15.48	5.91
3	Fibonacci	fibo.txt	305260	233193	567.70	579.83	7.82
4	Periodic	fss.txt	304118	281912	524.84	554.98	8.01
5	Protein	p1Mb.txt	1048576	69605	45.71	50.76	4.91
6	Protein	p2Mb.txt	2097152	139929	90.22	90.59	5.45
7	Random	random2.txt	510703	210122	377.93	370.62	8.64
8	Random	random21.txt	510703	24389	15.06	14.60	3.70
9	**Overall average**				**230.53**	**236.55**	**6.32**

McMaster University

Comparing speed performance per character of input



The results allow for a quick conclusion:

1. Overall, variant C is significantly faster than variants A and B. In fact by 3643%!
2. Even though variant A requires less additional memory, speed-wise does not do much better than B.
3. The speed of variants A and B is not proportional to the string's length. Rather, it mostly depends on the type of the string. It works better on strings with large alphabet size and low periodicity. This is intuitively clear, as for high periodicity strings the height of the search trees are large.

Memory saving modifications of Variant C

- C1: repetitions are collected for a round of K levels, then a sweep is executed and the resulting runs are reported, and the bucket memory is then reused in the next "batch" of repetitions. For our experiments, we used $K = 100$, so we refer to this variant as C1-100.
- C2: we consolidate repetitions with small periods ($\leq K$) into runs when putting them to the buckets (this saves memory since there are fewer runs than repetitions). For a repetition with period $p \leq K$ and start s, we check p buckets to the left and to the right of s; for $p > K$, we check K buckets to the left and to the right of s. This guarantees that all repetitions up to period K have been consolidated into runs before the final sweep, while repetitions of periods $> K$ are partially consolidated.

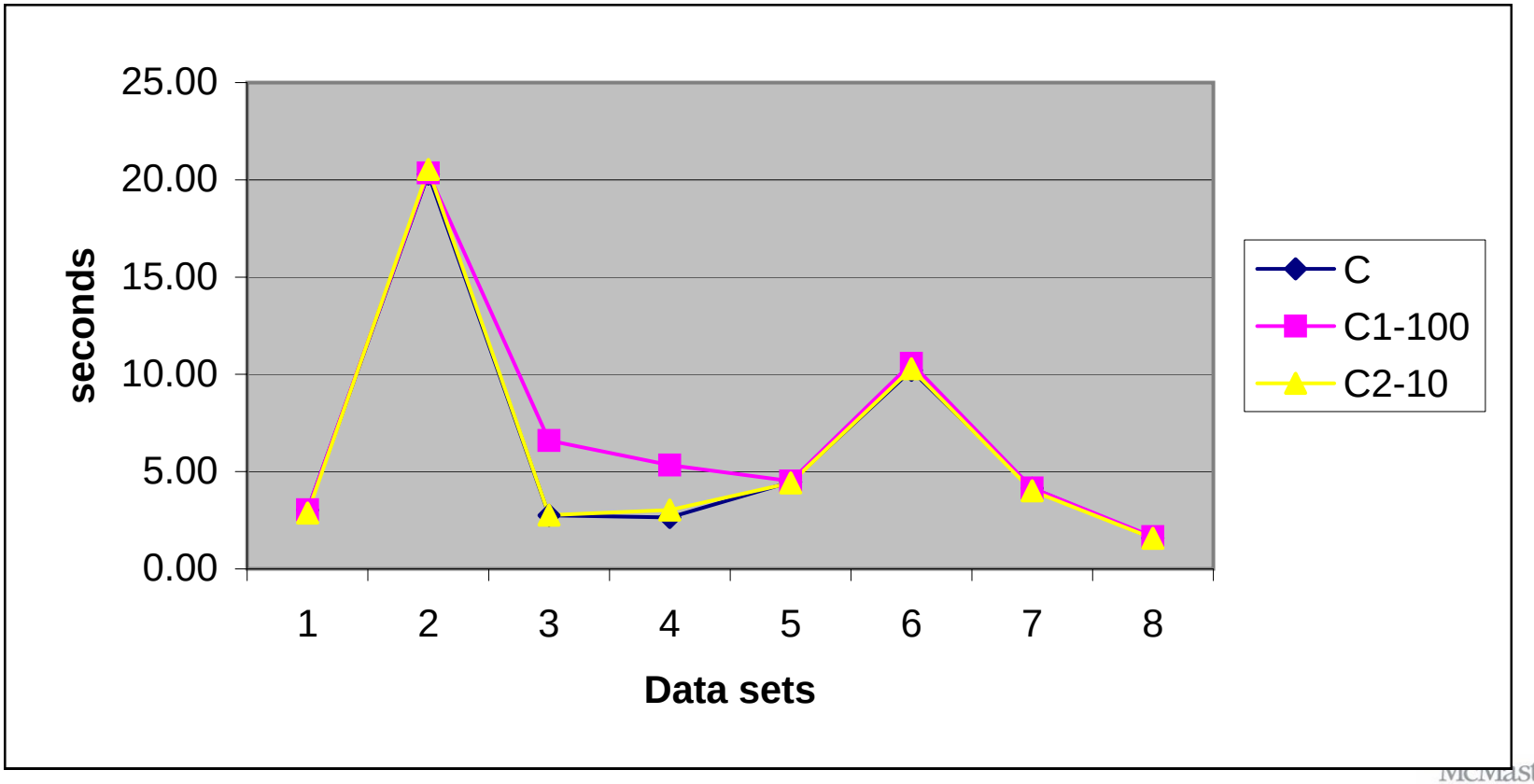
- Thus the final sweep ignores the repetitions with periods $\leq K$. Beside saving memory, the final sweep is a bit shorter, while putting repetitions into the buckets is a bit longer. For our experiments, we used $K = 10$, so we refer to this variant as C2-10.

Comparing speed performance of the variants C, C1-100, and C2-10

	Data Set	File Name	File size (bytes)	# of runs	Time (seconds)		
					C	C1-100	C2-10
1	DNA	dna.dna4	510976	130368	3.02	3.04	2.87
2	English	bible.txt	4047392	63690	20.29	20.36	20.53
3	Fibonacci	fibo.txt	305260	233193	2.75	6.60	2.76
4	Periodic	fss.txt	304118	281912	2.65	5.34	3.02
5	Protein	p1Mb.txt	1048576	69605	4.47	4.52	4.42
6	Protein	p2Mb.txt	2097152	139929	10.21	10.56	10.29
7	Random	random2.txt	510703	210122	4.15	4.16	4.01
8	Random	random21.txt	510703	24389	1.59	1.65	1.57

McMaster University

Comparing speed performance of the variants C, C1-100, and C2-10

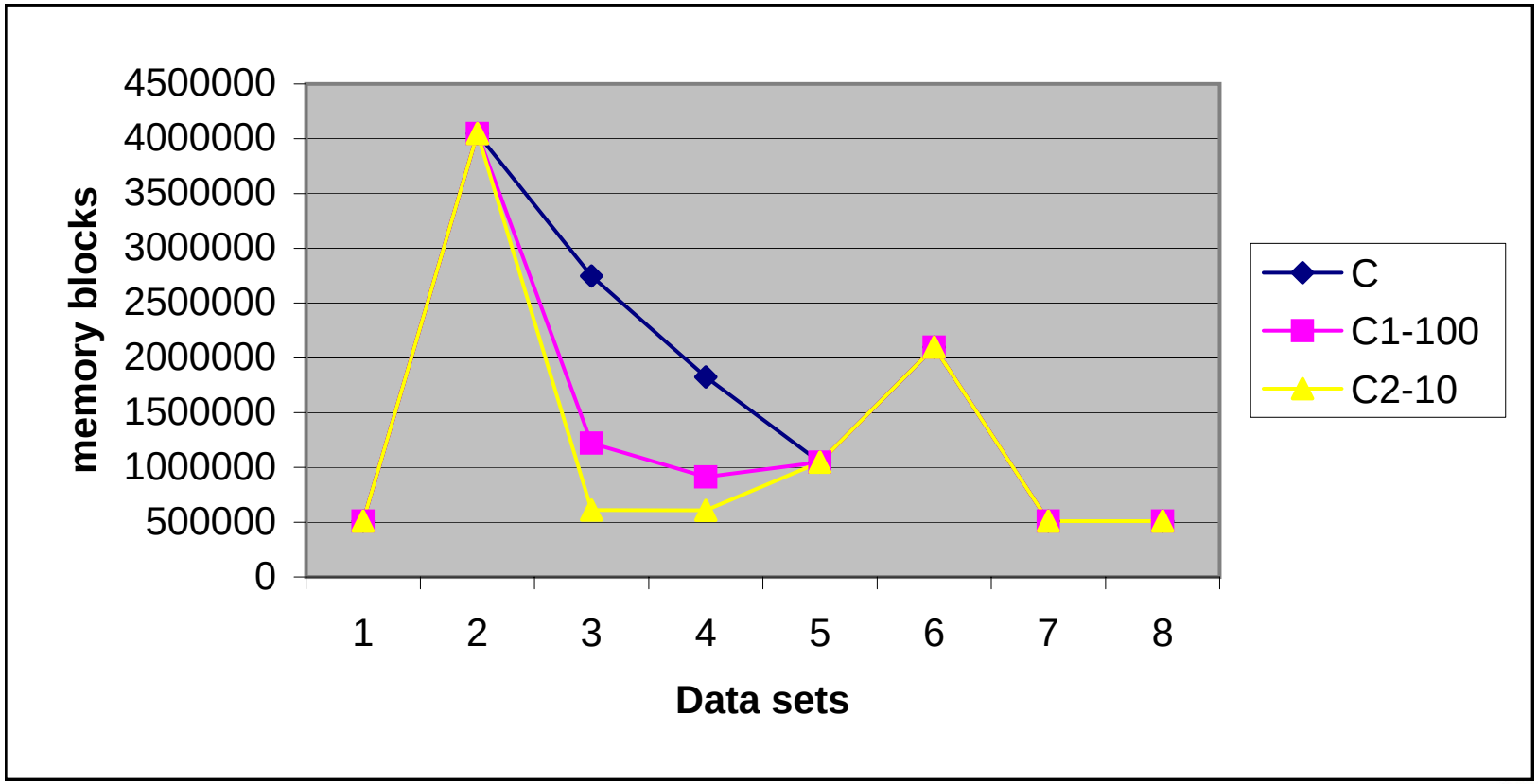


Comparing memory usage of the variants C, C1-100, and C2-10

	Data set	File name	File size (bytes)	Alphabet size	# of runs	Memory (blocks)		
						C	C1-100	C2-10
1	DNA	dna.dna4	510976	5	130368	510976	510976	510976
2	English	bible.txt	4047392	63	63690	4047392	4047392	4047392
3	Fibonacci	fibo.txt	305260	2	233193	2747340	1221040	610520
4	Periodic	fss.txt	304118	2	281912	1824708	912354	608236
5	Protein	p1Mb.txt	1048576	23	69605	1048576	1048576	1048576
6	Protein	p2Mb.txt	2097152	23	139929	2097152	2097152	2097152
7	Random	random2.txt	510703	2	210122	510703	510703	510703
8	Random	random21.txt	510703	21	24389	510703	510703	510703

McMaster University

Comparing memory usage of the variants C, C1-100, and C2-10



For the next set of tests we used large strings with large number of runs. The strings were obtained from W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara's website dedicated to "Lower Bounds for the Maximum Number of Runs in a String" at URL

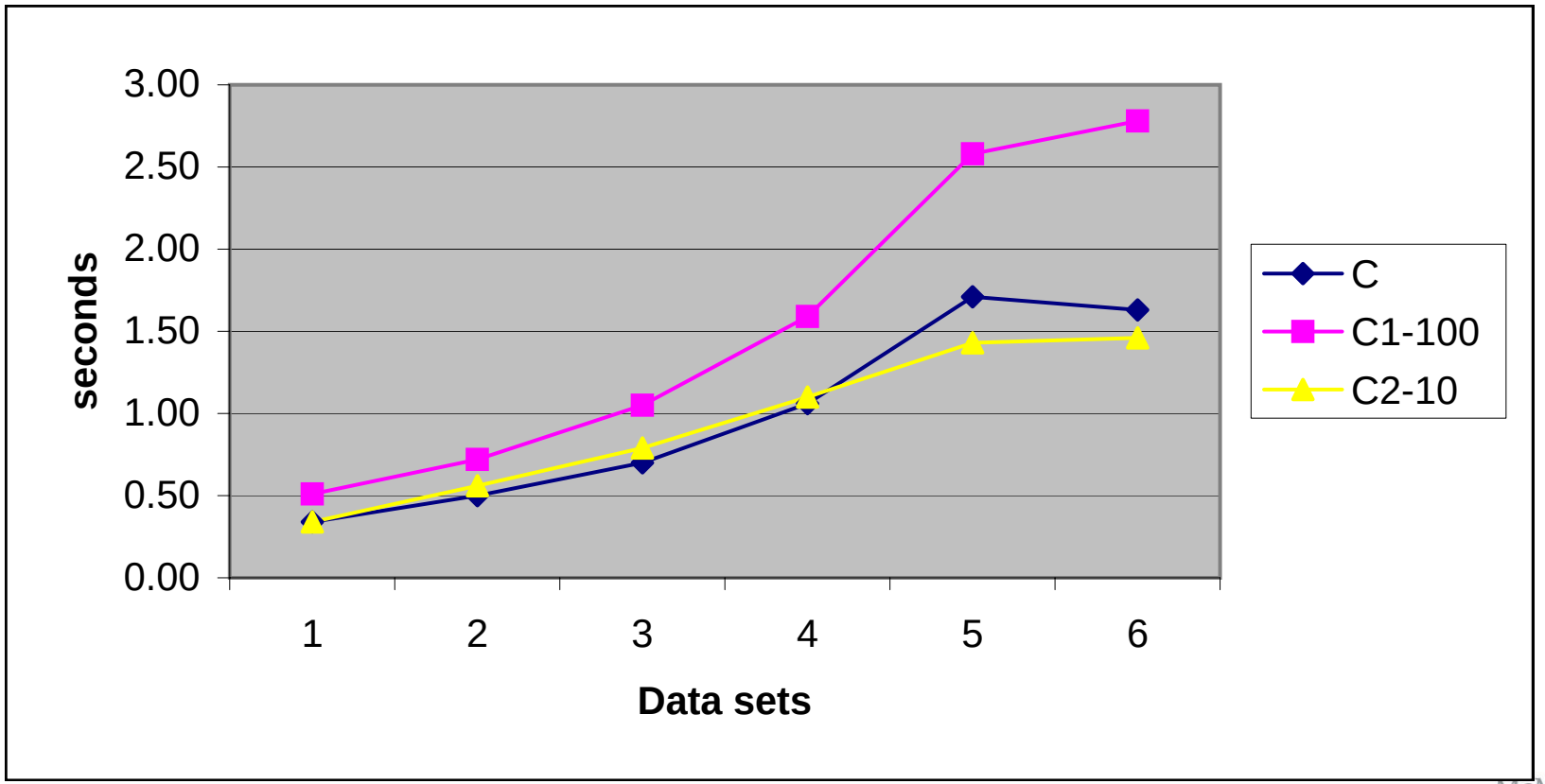
`http://www.shino.ecei.tohoku.ac.jp/runs/`

Comparing speed of C, C1-100, and C2-10 on large run-rich strings

	File name	File size (bytes)	Alphabet size	# of runs	Time (seconds)		
					C	C1-100	C2-10
1	60064.txt	60064	2	56714	0.34	0.51	0.34
2	79568.txt	79568	2	75136	0.50	0.72	0.56
3	105405.txt	105405	2	99541	0.70	1.05	0.79
4	139632.txt	139632	2	131869	1.06	1.59	1.10
5	176583.txt	176583	2	166772	1.71	2.58	1.43
6	184973.txt	184973	2	174697	1.63	2.78	1.46

McMaster University

Comparing speed of C, C1-100, and C2-10 on large run-rich strings

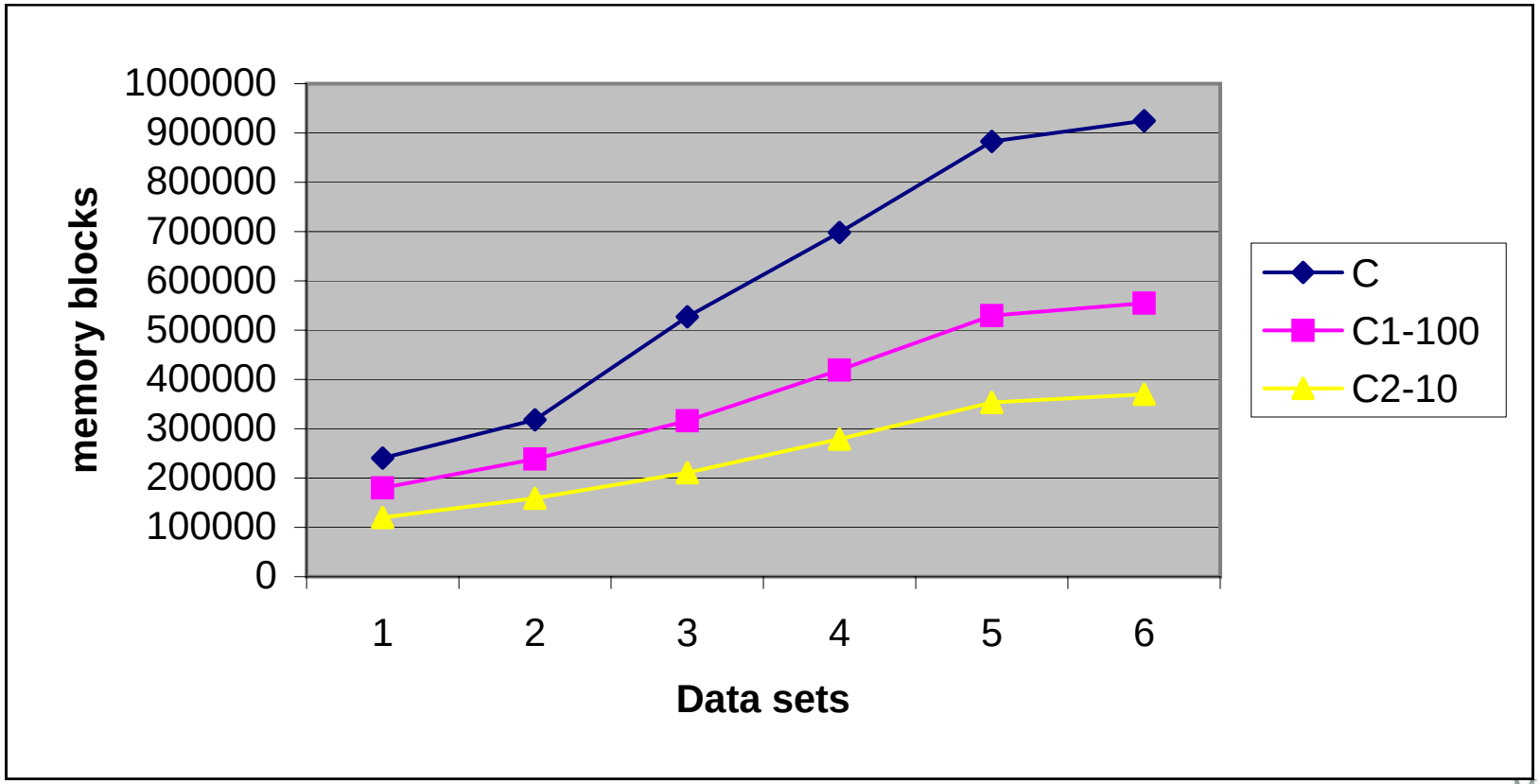


Memory usage of C, C1-100, and C2-10 on large run-rich strings

	File name	File size (bytes)	Alphabet size	# of runs	Time (seconds)		
					C	C1-100	C2-10
1	60064.txt	60064	2	56714	240256	180192	120128
2	79568.txt	79568	2	75136	318272	238704	159136
3	105405.txt	105405	2	99541	527025	316215	210810
4	139632.txt	139632	2	131869	698160	418896	279264
5	176583.txt	176583	2	166772	882915	529749	353166
6	184973.txt	184973	2	174697	924865	554919	369946

McMaster University

Memory usage of C, C1-100, and C2-10 on large run-rich strings



Conclusion

- We extended Crochemore's repetitions algorithm to compute runs.
- Of the three variants, variant C is by far more efficient time-wise, but requiring $O(n \log n)$ additional memory.
- However, its performance warranted further investigation into further reduction of memory requirements.
- The preliminary experiments indicate that C2-K is the most efficient version and so it is the one that should be the used as the basis for parallelization.
- Let us remark that variant C (and any of its modifications) could be used as an extension of any repetitions algorithm that reports repetitions of the same period together.

McMaster University

THANK YOU

McMaster University