A New approximate matching compression algorithm for DNA sequences

[a,b]J.M. Lázaro-Guevara*, [c] K.M. Garrido
jose.lazaroguevara@utah.edu, *dra.kmgarrido@gmail.com*

[a] Department of Human Genetics, University of Utah, Salt Lake City, USA
[b] Medical Sciences, University of San Carlos, Guatemala, Guatemala
[c] Paediatrics Department, Guatemalan Social Secure, *Guatemala, Guatemala*

* Corresponding Author.

1. Abstract: Undeveloped countries like Guatemala, where access to high-speed internet connections is limited, downloading and sharing Biological information of thousands of Mega Bits is a huge problem for the beginning and development of Bioinformatics. Based on that information is an urgent necessity to find a better way to share this biological data. There is when the compression algorithms become relevant. With all this information in mind, born the idea of creating a new algorithm using redundancy and approximate selection. **Methods:** Using the probability given by the transition matrix of the three-word tuple and relative frequencies. Calculating the relative and total frequencies given by the permutation formula (nr) and compressing 6 bits of information into 1 implementing the ASCII table code (0...255 characters, 28), using clusters of 102 DNA bases compacted into 17 string sequences. For decompressing, the inverse process must be done, except that the triplets must be selected randomly (or use a matrix dictionary, 4102). **Conclusion:** The compression algorithm has a better compression ratio than LZW and Huffman's algorithm. However, the time needed for decompressing makes this algorithm incompatible for massive data. The functionality as MD5sum need more research but is a promising helpful tool for DNA checking.

Keywords— DNA compression, DNA analysis, compression algorithm.

2. Background:

Undeveloped countries like Guatemala, where the access to high speed internet connections is limited downloading and sharing Biological information of thousands of Mega Bits is a huge problem for the beginning and development of Bioinformatics.

Based on that information is an urgent necessity to find a better way to share this biological data. There are two different ways to solve this problem, the first one improving the velocity at which the data is transferred (improving the internet connection), however this is not affordable in a country like Guatemala, the second way is to decrease the amount of data transferred.[1][2]

To accomplish this goal on reducing the data, the most common form is compressing the information, and for compressing the information there are several ways to do it.

But this compression forms depends on the kind of information or data to be compressed, the most reliable methods like Huffman Algorithm, cannot be used in the DNA compression because this methods based on relative frequencies can involve some data lost (~3%)[1]. A second way is using a more reliable method like **LZW (Lempel-Ziv-Welch)** where there is no lost on data but need the use of license to generate the compressed code based on patents, and involves the use of a matrix dictionary for each code compressed[2][3].

With all this information in mind, born the idea of creating a new algorithm based on redundancy and approximate probability given by the transition matrix of the three word tuple and relative frequencies [3][4](figure 1).

3. Methods:

3.1 Compressing Method:

First Step: Get a DNA sequence, and split it in subsequence of 102 bases

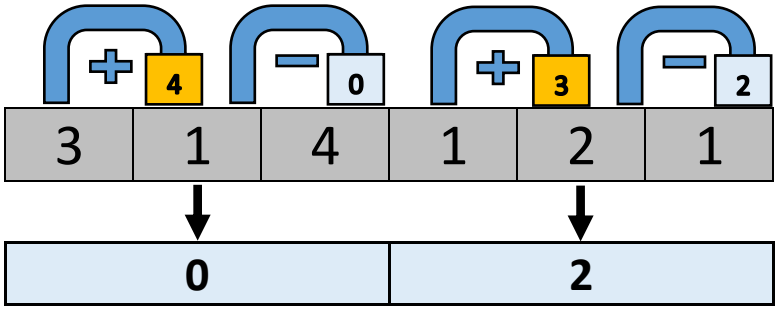
g	A	T	a	c	A

Second Step: transform the bases to a numeric correspondent (a=1, c=2, g=3, t=4).

3	1	4	1	2	1

Third Step: It is necessary to divide in 102 bases because is needed a number than can be divided by 6 and 3, and also superior a 100 to improve the compression ratio, based and that 1 character will represent a 6 DNA bases.

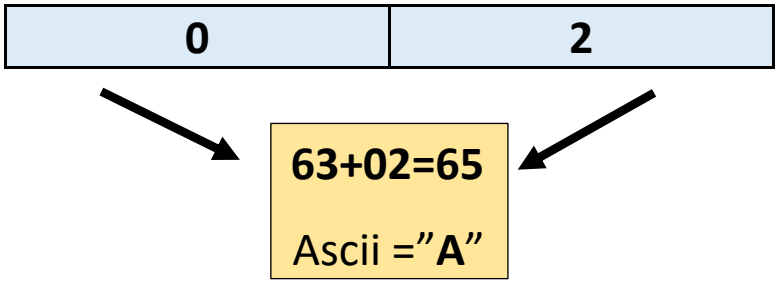
a.



b. Using permutation **n × n × … (r times) = n^r ,** r times, we have 4 possibilities (a,c,g,t), and selecting 3 each time, based on the process of adding the first one and subtracting to that sum the third value (eg. (3+1)-4=0), we only have 7 possible values that will be from **-2 to 7**. This model of adding and subtracting was used to obtain an array of 10 elements in a normal distribution (1/64,3/64,6/64,10/64,12/64) (table 1).

c. It is necessary to group 64 possible combination of values in only 10 combinations to use the ASCII mode (range from 0 to 255), in table I it is possible to observe that exist two negative values (-2,-1), and it is necessary to replace them in the second value by the (8 = -1, and 9 = -2), to use the concatenation of two characters as one number.

d. After obtaining 2 characters and concatenating it, it is necessary to add 63 to this value, to avoid characters used by the core system. The range of values after concatenation will go from -29 to 79. So after adding 63 to this values the range in ASCII will be from 34 to 142. Table II.



Example 1:

DNA code: (102 characters)

a t a a c c a c a a a g a g c a g g t a a c g a g c c c g c a g c t t g t a a g c g c g a g c t a t t a a c a
a t c g t t c c a c t t g g c a a c g c t a c c a g g a t g t t a a c a a a g t c c a t a g g

Compressed Version: (17 characters)

h[Tk`XihI*oi`?]N4

e. Checking numbers 1 and 2: This numbers allow to calculate the original 102 bases compacted into 17 characters from 34 numbers. Number 1 is obtained by adding and subtracting the product of the 34 calculated number. (eg. (3+1-4)-(1+2-1)….+…..-). In case of example 1 is **11.**

   Number 2 is obtained by a similar process using multiplication and subtracting process calculated number of the 34 by the formula (2*(1st*3rd))-(3*2nd) (eg. (2(3*4)-(3*1))-(2(1*1)-(3*2)….+…..-). **In case of example 1 is -3.**

   Compacted code of example 1: **h[Tk`XihI*oi`?]N4!11!-3**

f. Line 1 header. This is the first line of the compacted code and contains the absolute frequency of each one of the 64 possible combinations for future checking in the decompressing process. **(eg. 62-95-94-59-06-34-59-46-27-62-66-60-59-96-43-63-36-16-62-56-43-64-26-44)**

3.2 Decompressing Method:

a. Read Line 1, introduce the data into a linear vector with relative frequencies of each triplet for further comparison against the decompressed code and the original code, and improving the velocity of the change based on the change in the total amount of probability of each possible term.
b. Obtain the compacted code (e.g **h[Tk`XihI*oi`?]N4!11!-3**) , split into three different terms using the symbol "**!",** as separation indicator**. Term 1:** 17 character code. **Term 2:** number 1 and **Term 3:** number 2.
c. Using the reverse method of part d in compression method, convert the ascii code into a two digit value (eg. Asci = "A", number from asci = 65), subtracts **63** to that number (eg. 65 - 63=02), using the modulus and the integer division separate into 2 numbers (02 = 0, 2). If number two is 8 or 9 change into the negative portion (eg. **– 29 = – 2, – 2**).
d. From the 17 character code, after changing in the 2 digits way, it will be converted into a 34 digits code.

e. Now is necessary to change 34 digits into 102 characters, but it is no possible to use the inverted process on step b on the codifying part because each number has may possibilities from 3 to 12 possible options, only numbers -2 and 7, has one direct transcription, and number -2 and 7, where selected based on the most common triplet of bases repeated in 1000 trials of codifying sequences.

f. Because there are 102 possible DNA bases in this code and there are 4 possible letter for sequence and the order it is important, it is necessary to calculate the permutation **4 P 102**, that it is equivalent to count 34 digits and 64 possible values obtained from the triplets, **64 P 34** .

But this value is so huge (2.57 x 10E61), but it is limited as shown in **TABLE III**, by the number of probabilities from 3 to 12, in 8 possible numbers.
For that reason is necessary the checking numbers 1 and 2, there is only one sequence of 34digits (17 characters) for each pair of checking numbers (look at probability table at the bottom of TABLE III, figure 1).

3.3 Measuring Accuracy on decompression and data loss:

For obtaining the original sequence of DNA it is necessary to obtain the permutations, and compare it to the checking numbers, the problem is that this take a lot of time so the best solution is to import a common Tables Comparing Dictionary, the disadvantage of this solution is the big size of the dictionary.

The measuring of accuracy is related to the number of cycles permuted, to obtain the original DNA sequence is necessary compare checking numbers 1 and 2, until obtaining the original sequence, more the numbers of permutation higher the probability on obtaining the 102 original DNA bases per permutation. (TABLE IV)

The accuracy will be measured comparing simultaneously the sequences (Compressed = original), and measuring the differences or sections with discrepancy it is obtained the accuracy proportion (e.g)

Original	1	4	1	1	3	3	1	2	1	2	1	4	Sum	Sum/12 (%)
Decompressed	1	4	1	1	2	2	1	2	1	1	1	3		
Match	1	1	1	1	0	0	1	1	1	0	1	0	8	67%

4. Results:

After several trials, and running multiple simulations in R (Statistical Programming Language), on how this compression and decompression system works. (Figure 2, figure 3) It is obtained a compression ratio based on data weight in kilobytes (Kb), using a 10,000 bases (A, C, G, T) section of DNA extracted from NC_000913.1 (E.coli) with read.Genbank function.

The data extracted and printed in a text file (.TXT), uses 20,100 bytes, the data compacted in the same archive system uses only 2,567.

The definition of the compression ratio is (|O|/2| I|), where |I| is number of bases in the input DNA sequence and |O| is the length (number of bits) of the output sequence, $\text{Compression Ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}$, **the general compression ratio is 8:1** (7.84), the **inline code compression ratio is 5:1** (102 DNA bases are equivalents to 17 ASCII codes plus 2 checking numbers). The compression ratios, as well as the multiple permutation system, are presented in Figure 1.

The compressing and decompressing time is related to the number of cycles (~102 permutations), using a quad-core 2.3 Ghz processor as standard, it takes **38 milliseconds** to complete one cycle, but because it is using a iteration systems (for and while commands), taking on count the RAM and processor bus transfer speed is necessary to add 15% on delay each 1000 cycles.

The accuracy on decoding is related directly to the number of iterations and permutations (figure 2), whit only 100 to 2,500 accuracy varies from 50 to 70%, but this accuracy is counted only in a line of decoding (102 DNA bases), based on time of processing, after 7,500 cycles accuracy increase in exponential, in range from 80 to 91%, reaching the limit in cycles around 50,000, because Random Access Memory limiting, so it is unknown if the accuracy will reach 100% in the estimated 450,000 cycles (RAM memory insufficient for this calculation on normal PC 8 Gigabytes).

After running a sequence of 1020 DNA bases, using 5000 cycles, the analysis took 9.5 hours, and accuracy obtained was 63%. A combination dictionary for improving the speed and accuracy was created with 120,000 permutations and speed increase by 5%, for creating the complete dictionary with all possible permutations (2.57 x 10E61), certainly will improve accuracy and speed of decoding, but was not possible to create it with the R programming language, based on the RAM size.

5. Discussion:

This algorithm have demonstrated that the general compression ratio obtained (8:1) is better that traditional compression codes (eg LZW, Huffmans, Biocompress, Gencompress, FASTA), but the time needed for decompression makes no worthy the usage of this compression method by using multiple permutations, it will improve using a dictionary of sequences preinstalled in the coding and decoding software.

It was not possible to probe the complete accuracy, assuring no data lose in the decompression system after 50,000 cycles accuracy only reach 86%, adding some modifications like improving the random sampling permutations it was able to obtain 91%. For that reason is necessary to do more changes on the code and try to reach 100% accuracy on selected sampling instead of random sampling on a better equipment with more processor speed and RAM.

Besides the time consumption for decompressing, this algorithm can also be used in the compression form as a checksum like the MD5sum system for DNA codes. MD5 algorithm is the mainstream for the cryptographic check and file checkup, the hash function when MD5 is applied returns a unique check number that allow knowing that this file is uncorrupted, this is an important problem in compressing method when data lose exist, and is necessary to be sure of that DNA is not corrupted.

So it is necessary to continue in researching for improving it.

6. Conclusion:

- The compression algorithm has better compression ratio than LZW and huffmans algorithm.

- The time need for decompressing make not useful this algorithm for huge amount of data.

- The functionality as MD5sum need more research, but is a promising useful tool for DNA checking.

7. Reference:

[1]. Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. C 1990. Second Edition

[2] Welch, T.A., "A Technique for High-Performance Data Compression," in Computer , vol.17, no.6, pp.8-19, June 1984 doi: 10.1109/MC.1984.1659158

[3] A compression algorithm for DNA sequences DOI: 10.1109/51.940049

[4] Hsi-Yang Fritz, Markus et al. “Efficient Storage of High Throughput DNA Sequencing Data Using Reference-Based Compression.” Genome Research21.5 (2011): 734–740. PMC. Web. 3 Dec. 2015.

[5] Zhao Yong-Xia; Zhen Ge, "MD5 Research," in Multimedia and Information Technology (MMIT), 2010 Second International Conference on , vol.2, no., pp.271-273, 24-25 April 2010 doi: 10.1109/MMIT.2010.186

TABLE I

64 permutations in Triplets for DNA encoding and d ecoding

(Numerical representation of DNA Triplets)

Replace	Combination Values				Distribution		Combination Values				Distribution	
9	-2	1	1	4	1	1/64	3	2	4	3		
8	-1	1	2	4			3	1	4	2		
8	-1	2	1	4	1/3	3/64	3	3	4	4		
8	-1	1	1	3			3	4	2	3		
	0	2	2	4			3	2	2	1		
	0	1	2	3			3	3	2	2		
	0	2	1	3	1/6	6/64	3	4	1	2	1/12	12/64
	0	1	1	2			3	3	1	1		
	0	3	1	4			3	4	3	4		
	0	1	3	4			3	2	3	2		
	1	1	4	4			3	1	3	1		
	1	2	2	3			3	3	3	3		
	1	1	2	2			4	4	4	4		
	1	3	2	4			4	2	4	2		
	1	4	1	4			4	1	4	1		
	1	2	1	2	1/10	10/64	4	3	4	3		
	1	1	1	1			4	4	2	2	1/10	10/64
	1	3	1	3			4	3	2	1		
	1	2	3	4			4	4	1	1		
	1	1	3	3			4	4	3	3		
	2	2	4	4			4	2	3	1		
	2	1	4	3			4	3	3	2		
	2	4	2	4			5	4	4	3		
	2	2	2	2			5	2	4	1		
	2	1	2	1			5	3	4	2		
	2	3	2	3			5	4	2	1	1/6	6/64
	2	4	1	3	1/12	12/64	5	4	3	2		
	2	2	1	1			5	3	3	1		
	2	3	1	2			6	4	4	2		
	2	2	3	3			6	3	4	1	1/3	3/64
	2	1	3	2			6	4	3	1		
	2	3	3	4			7	4	4	1	1	1/64

TABLE II

ASCII SEQUENCE FOR CHAR AND HEXADECIMAL COMPRESION

A) Primary ASCII code (2^7)

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

B) Extended ASCII Code (2^8)

128	Ç	144	É	160	á	176	▒	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	|	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	n
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	2
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

TABLE III

Sequence of DNA of Example 1, possible results on checking numbers 1 and 2 after permutations.

	4	1	2	-1	2	1	4	4	3	3	2	5	4	2	4	1	1	0	-2	1	4	-1	4	2	3	3	0	0	3	0	1	5	-1	1
1	444	144	244	124	244	144	444	444	243	243	244	443	444	244	444	144	144	224	114	144	444	124	444	244	243	243	224	224	243	224	144	443	124	144
2	242	223	143	214	143	223	242	242	142	142	143	241	242	143	242	223	223	123		223	242	214	242	143	142	142	123	123	142	123	223	241	214	223
3	141	122	424	113	424	122	141	141	344	344	424	342	141	424	141	122	122	213		122	141	113	141	424	344	344	213	213	344	213	122	342	113	122
4	343	324	222		222	324	343	343	423	423	222	421	343	222	343	324	324	112		324	343		343	222	423	423	112	112	423	112	324	421		324
5	422	414	121		121	414	422	422	221	221	121	432	422	121	422	414	414	314		414	422		422	121	221	221	314	314	221	314	414	432		414
6	321	212	323		323	212	321	321	322	322	323	331	321	323	321	212	212	134		212	321		321	323	322	322	134	134	322	134	212	331		212
7	411	111	413		413	111	411	411	412	412	413		411	413	411	111	111			111	411		411	413	412	412			412		111			111
8	433	313	211		211	313	433	433	311	311	211		433	211	433	313	313			313	433		433	211	311	311			311		313			313
9	231	234	312		312	234	231	231	434	434	312		231	312	231	234	234			234	231		231	312	434	434			434		234			234
10	332	133	233		233	133	332	332	232	232	233		332	233	332	133	133			133	332		332	233	232	232			232		133			133
11			132		132				131	131	132			132										132	131	131			131					
12			334		334				333	333	334			334										334	333	333			333					

TABLE IV

Sequence of DNA of Example 1, checking numbers 1 and 2.

	4	1	2	-1	2	1	4	4	3	3	2	5	4	2	4	1	1	0	-2	1	4	-1	4	2	3	3	0	0	3	0	1	5	-1	1
1	20	-4	4	2	4	-4	20	20	0	0	4	12	20	4	20	-4	-4	10	5	-4	20	2	20	4	0	0	10	10	0	10	-4	12	2	-4
2	-4	6	-6	13	-6	6	-4	-4	-8	-8	-6	-8	-4	-6	-4	6	6	0		6	-4	13	-4	-6	-8	-8	0	0	-8	0	6	-8	13	6
3	-10	-2	26	3	26	-2	-10	-10	12	12	26	0	-10	26	-10	-2	-2	9		-2	-10	3	-10	26	12	12	9	9	12	9	-2	0	3	-2
4	6	18	2		2	18	6	6	18	18	2	2	6	2	6	18	18	1		18	6		6	2	18	18	1	1	18	1	18	2		18
5	10	29	-4		-4	29	10	10	-2	-2	-4	7	10	-4	10	29	29	21		29	10		10	-4	-2	-2	21	21	-2	21	29	7		29
6	0	5	12		12	5	0	0	6	6	12	-3	0	12	0	5	5	-1		5	0		0	12	6	6	-1	-1	6	-1	5	-3		5
7	5	-1	21		21	-1	5	5	13	13	21		5	21	5	-1	-1			-1	5		5	21	13	13			13		-1			-1
8	15	15	1		1	15	15	15	3	3	1		15	1	15	15	15			15	15		15	1	3	3			3		15			15
9	-5	7	9		9	7	-5	-5	23	23	9		-5	9	-5	7	7			7	-5		-5	9	23	23			23		7			7
10	3	-3	3		3	-3	3	3	-1	-1	3		3	3	3	-3	-3			-3	3		3	3	-1	-1			-1		-3			-3
11			-5		-5				-7	-7	-5			-5										-5	-7	-7			-7					
12			15		15				9	9	15			15										15	9	9			9					
	11	-32																																

FIGURE 1

M	A	L	T	K	A	E	M	S	E	Y	L	F	...
ATG	GCG	CTT	ACA	AAA	GCT	GAA	ATG	TCA	GAA	TAT	CTG	TTT	...
1.000	0.469 0.057 0.275 0.199	0.018 0.018 0.038 0.033 0.007 0.888	0.451 0.468 0.035 0.046	0.798 0.202	0.469 0.057 0.275 0.199	0.794 0.206	1.000	0.428 0.319 0.033 0.007 0.037 0.176	0.794 0.206	0.193 0.807	0.018 0.018 0.038 0.033 0.007 0.888	0.228 0.772	
ATG	GCT GCC GCA GCG	TTA TTG CTT CTC CTA CTG	ACT ACC ACA ACG	AAA AAG	GCT GCC GCA GCG	GAA GAG	ATG	TCT TCC TCA TCG AGT AGC	GAA GAG	TAT TAC	TTA TTG CTT CTC CTA CTG	TTT TTC	

Imaged copied from Slides on Statistical Methods in Genetics and Bioinformatics MA584 at WPI

FIGURE 2

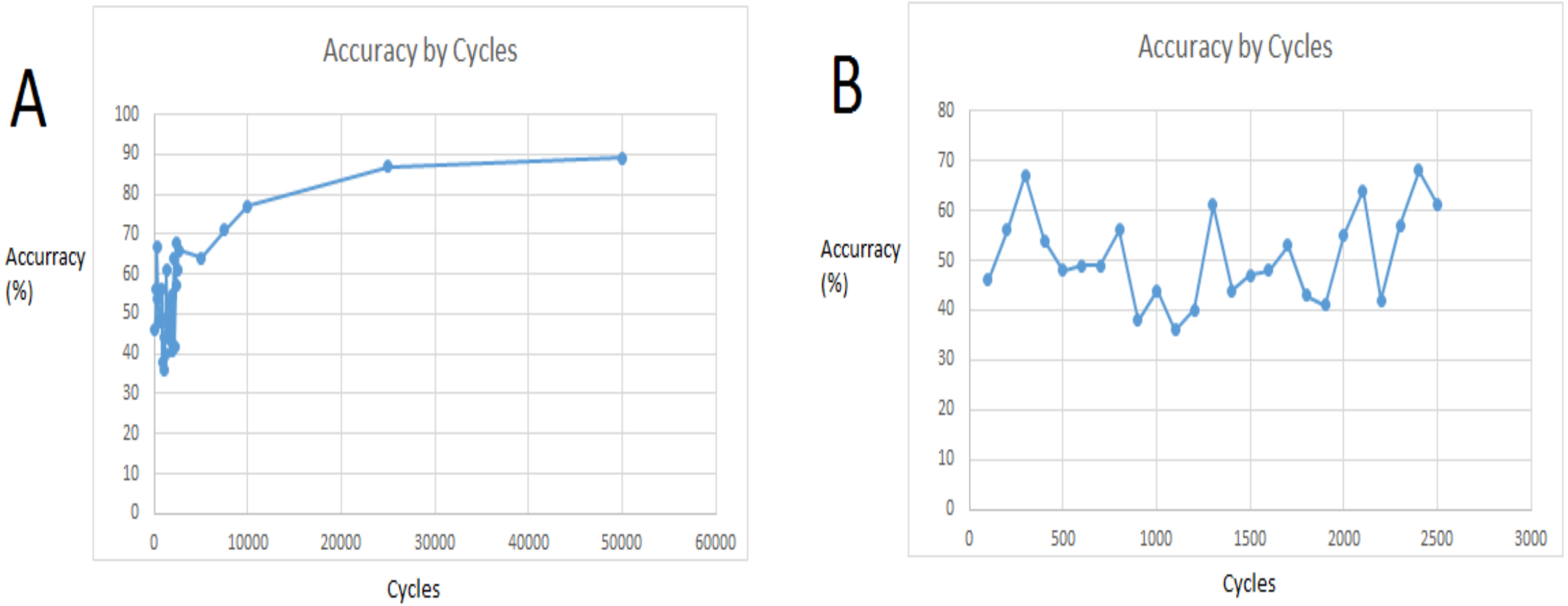
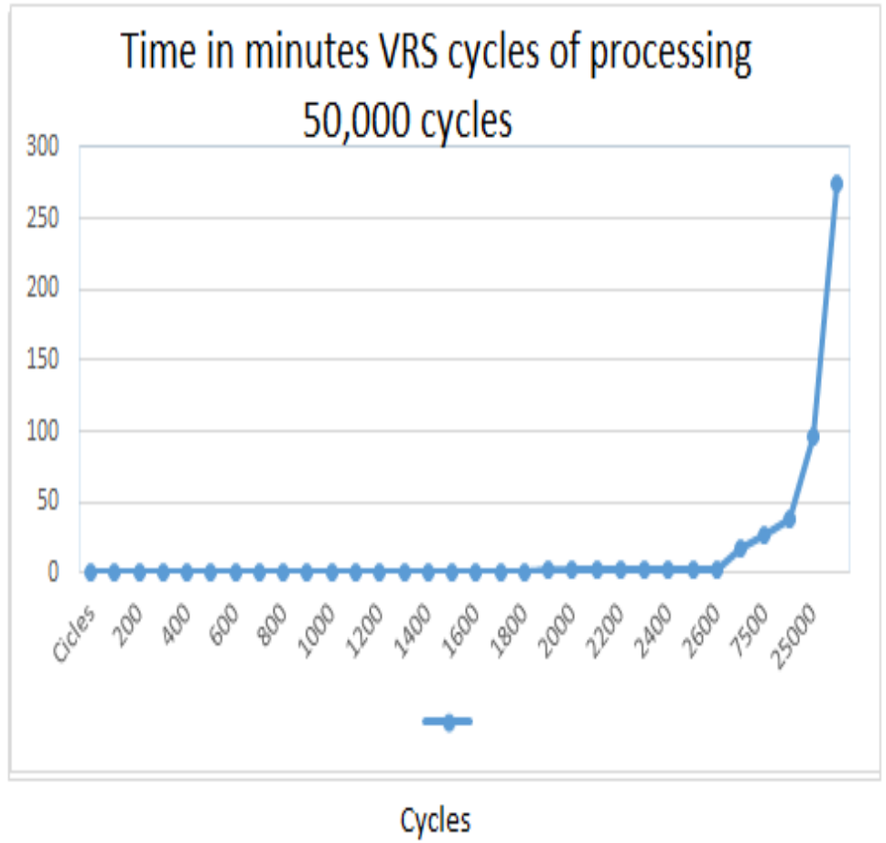
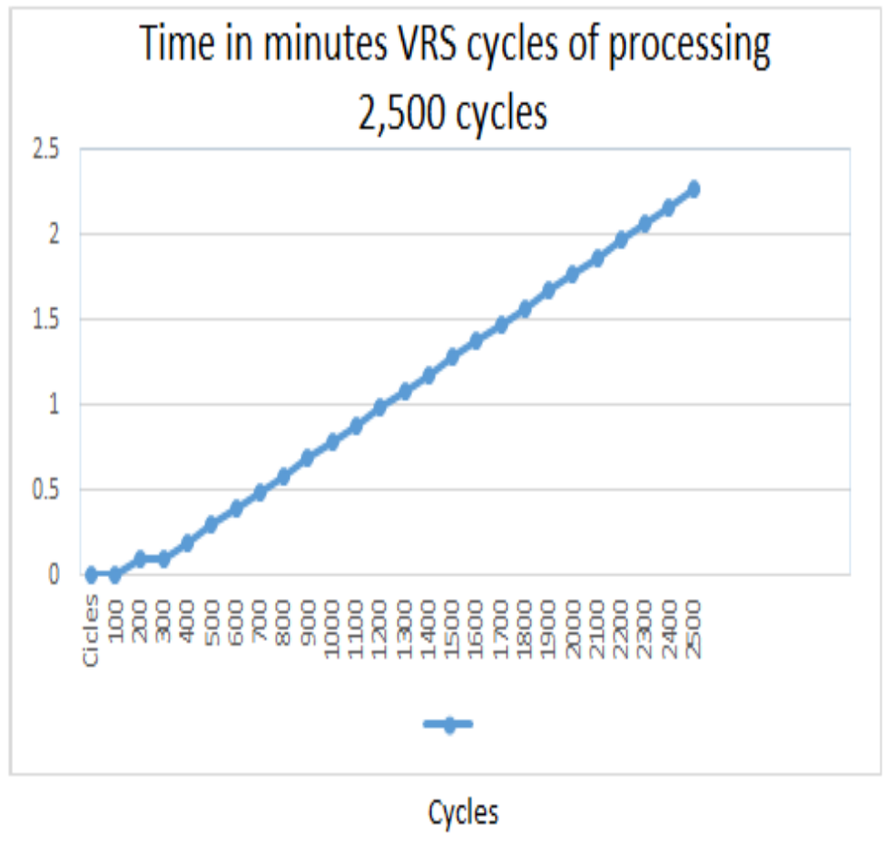

A
Accuracy by Cycles
Accurracy (%)
100
90
80
70
60
50
40
30
20
10
0
0
10000
20000
30000
40000
50000
60000
Cycles
B
Accuracy by Cycles
Accurracy (%)
80
70
60
50
40
30
20
10
0
0
500
1000
1500
2000
2500
3000
Cycles

FIGURE 3

A


Time in minutes VRS cycles of processing
50,000 cycles
Minutes
300
250
200
150
100
50
0
Cicles
200
400
600
800
1000
1200
1400
1600
1800
2000
2200
2400
2600
7500
25000
Cycles

B


Time in minutes VRS cycles of processing
2,500 cycles
Minutes
2.5
2
1.5
1
0.5
0
Cicles
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
2200
2300
2400
2500
Cycles

Compressing Code in R

```
#library(ape);
library(stringr);
library(gtools)
#Program: DNA coding.
ThreeWordSeq = function(seq)
{
x=c("A","C","G","T");
valores =permutations(n=4,r=3,v=x,repeats.allowed=T);

seq3=getNumSeq(seq);
tripletas=rep(0,64);
return(tripletas);
j=1;
while(j<=length(seq))
	{
	char=paste0(seq[i],seq[i+1],seq[i+2]);
	char=toupper(chr);
	if(suma==350){valor=76}

	j=j+3;
	}
}
TwoWordSeq = function(seq)
	{
	largo=length(seq);
	largo=largo-1;
	AA=0;CA=0;GA=0;TA=0;AC=0;CC=0;GC=0;TC=0;
AG=0;CG=0;GG=0;TG=0;AT=0;CT=0;GT=0;TT=0;
	for (i in 1:largo)
		{
		chr=paste0(seq[i],seq[i+1]);
		chr=toupper(chr);
			if (chr=="AA") {
				AA = AA+1;
			}else if (chr=="AC") {
				AC=AC+1;
			}else if (chr=="AG") {
				AG=AG+1;
			}else if (chr=="AT") {
				AT=AT+1;
			}else if (chr=="CA") {
				CA=CA+1;
			}else if (chr=="CC") {
				CC=CC+1;
			}else if (chr=="CG") {
				CG=CG+1;
			}else if (chr=="CT") {
				CT=CT+1;

			}else if (chr=="GA") {
				GA=GA+1;
			}else if (chr=="GC") {
				GC=GC+1;
			}else if (chr=="GG") {
				GG=GG+1;
			}else if (chr=="GT") {
				GT=GT+1;
			}else if (chr=="TA") {
				TA=TA+1;
			}else if (chr=="TC") {
				TC=TC+1;
			}else if (chr=="TG") {
				TG=TG+1;
			}else if (chr=="TT") {
				TT=TT+1;
			}
		}
TwoWord=c(AA,AC,AG,AT);
TwoWord=c(TwoWord,c(CA,CC,CG,CT));
TwoWord=c(TwoWord,c(GA,GC,GG,GT));
TwoWord=c(TwoWord,c(TA,TC,TG,TT));
	return(TwoWord);
	}

#simulated sequence using iid model
x= c("a","c","g","t");
seq2 = sample(x, 10000, replace=TRUE);
TwoWordMatrix = TwoWordSeq(seq2);

	#Function transfer DNA seq to numerical: a=1; c=2;
g=3; t=4;
	#Input: a sequence (array) of "a", "c", "g", "t"
	#Output: the corresponding numeric values
	getNumSeq = function(seq) {
		numericalSeq = array(0, length(seq));
		for (i in 1:length(seq)) {
			if (seq[i]=="a") {
				numericalSeq[i] = 1;
			}else if (seq[i]=="c") {
				numericalSeq[i] = 2;
			}else if (seq[i]=="g") {
				numericalSeq[i] = 3;
			}else if (seq[i]=="t") {
				numericalSeq[i] = 4;
			}
		}
		return(numericalSeq);
	}
	#source();

is.even <- function(x) x %% 2 == 0
is.odd <- function(x) x %% 2 != 0

codificador = function(sequency)
{
contador=0;
calibrador=rep(0,34);
```

```
sumatoria=rep(0,34);
codificacion=rep("0",21);
codificacion[18]="!";
codificacion[20]="!";

j=1;

while (j<=length(sequency))
        {
        contador=contador+1;
        sumatoria[contador]=strtoi(sequency[j])+strtoi(sequ
ency[j+1])-strtoi(sequency[j+2]);
        calibrador[contador]=(2*(strtoi(sequency[j])*strtoi(se
quency[j+2])));
    calibrador[contador] =calibrador[contador] -
(3*strtoi(sequency[j+1]));
        j=j+3;
        }

valor=0;
contador=0;
j=1;
while (j <=length(sumatoria))
        {
        x=sumatoria[j]
        y=sumatoria[j+1]
        if(y==-2)
                {
                y=9;
                }
        if(y==-1)
                {
                y=8;
                }
        valor=strtoi(paste0(x,y))
        valor=valor+63;
        contador=contador+1;
        codificacion[contador]=chr(valor);
        j=j+2;
        }
valor=0;
valor2=0;
for (i in 1:length(sumatoria))
        {
        if(is.odd(i)==TRUE)
                {
                valor=valor+sumatoria[i];
                valor2=valor2+calibrador[i];
                }
        else
                {
                valor=valor-sumatoria[i];
                valor2=valor2-calibrador[i];
                }
        }

codificacion[19]=toString(valor);
codificacion[21]=toString(valor2);
return(codificacion);
}

#based on the first letter is the missing code (sum letter)
.....
#for example if the code is 1000 we only will have 999, but
the missin letter
#is on the first letter of the code.

write(seq2,"Datos.txt",100);
datos = scan("Datos_check.txt", what=character(1))
seq1=TwoWordSeq(datos);
splited = c(seq1[1],"-",seq1[2],"-",seq1[3],"-",seq1[4],"-
",seq1[5],"-",seq1[6],"-",
seq1[7],"-",seq1[8],"-",seq1[9],"-",seq1[10],"-",seq1[11],"-
",seq1[12],"-",
seq1[13],"-",seq1[14],"-",seq1[15],"-",seq1[16]);
write(c(datos[1],"-",splited),"compacted.txt",100,sep="");

contador = 0;
line=0;
ExtractoSeq = rep(0,102);
for (i in 1:length(datos))
        {
        contador=contador+1;
        ExtractoSeq[contador]=datos[i];
        if ((contador==102)||(i==length(datos)))
                {
                line=line+1;
                contador=0;
                seq2=getNumSeq(ExtractoSeq);
                codificado=rep("0",17);
                codificado=codificador(seq2);

        write(c(codificado),"compacted.txt",1024,append =
TRUE,sep="");
                ExtractoSeq = rep(0,102);
                }
        }
```

Decompressing Code in R

```
#library(ape);
#library(combinat);
library(stringr);
library(gtools)
#Program: DNA coding.
TransMatrix=rep(0,16);

is.even <- function(x) x %% 2 == 0
is.odd <- function(x) x %% 2 != 0

lectura = function(data)
{
	sub1=word(data[2],1,sep="!");
	sub2=word(data[2],2,sep="!");
	sub3=word(data[2],3,sep="!");
	decodificada=c(sub1,sub2,sub3);

for (i in 3:length(data))
	{
	sub1=word(data[i],1,sep="!")
	sub2=word(data[i],2,sep="!")
	sub3=word(data[i],3,sep="!")
	TransMatrix[i]=strtoi(substring);
	decodificada=c(decodificada,c(sub1,sub2,sub3))
	}
return(decodificada);
}
########
numerifica = function(data)
{
largo=dim(data)[1];
ancho=dim(data)[2];
Transision=matrix(0,nrow=largo,ncol=36);
for (i in 1:largo)
	{
	for (j in 1:17)
		{
		valor=asc(data[i,j]);
		valor=strtoi(valor)-63;
		suma=sum(valor);
		if (length(valor)==1)
			{

	valor1=ifelse(valor>=0,valor%/%10,(valor*-
1)%/%10);

	valor2=ifelse(valor>=0,valor%%10,(valor*-
1)%%10);

	valor1=ifelse(valor>=0,valor1,(valor1*-1));

				valor2=ifelse(valor2==8,-1,valor2);
				valor2=ifelse(valor2==9,-2,valor2);
				Transision[i,((j*2)-1)]=valor1;
				Transision[i,((j*2))]=valor2;
				}
			else
				{
				if(suma==197){valor=66}
				if(suma==209){valor=78}
				if(suma==211){valor=73}
				if(suma==217){valor=77}
				if(suma==218){valor=68}
				if(suma==231){valor=75}
				if(suma==319){valor=67}
				if(suma==323){valor=69}
				if(suma==325){valor=71}
				if(suma==326){valor=72}
				if(suma==331){valor=70}
				if(suma==339){valor=65}
				if(suma==341){valor=74}
				if(suma==350){valor=76}

	#print(c(probability[i,j],"suma:",sum(valor)))

	valor1=ifelse(valor>=0,valor%/%10,(valor*-
1)%/%10);

	valor2=ifelse(valor>=0,valor%%10,(valor*-
1)%%10);

	valor1=ifelse(valor>=0,valor1,(valor1*-1));

				if(valor2==8){valor2=-1}
				if(valor2==9){valor2=-2}
				Transision[i,((j*2)-1)]=valor1;
				Transision[i,((j*2))]=valor2;
				}
		}

	Transision[i,35]=strtoi(data[i,18]);
	Transision[i,36]=strtoi(data[i,19]);
	}

return(Transision);
}
###
buscar = function(numero)
{
x=c(1,2,3,4);
valores =permutations(n=4,r=3,v=x,repeats.allowed=T);
tripleta2=rep(0,3);
suma=rep(0,64);
valores=cbind(valores,suma);
valores=cbind(valores,suma);
for (i in 1:64)
	{
	valores[i,4]=valores[i,1]+valores[i,2]-valores[i,3];
	valores[i,5]=((2*(valores[i,1]*valores[i,3]))-
(3*valores[i,2]));
	}
```

```
for (i in 1:64)
	{
	if(numero==valores[i,4])
		{

	tripleta1=c(valores[i,1],valores[i,2],valores[i,3]);
		###print(tripleta1)
		tripleta2=rbind(tripleta2,tripleta1);
		}
	}
largo=dim(tripleta2)[1];
x3=sample(2:largo,1);
cadenab=tripleta2[x3,];
return(cadenab);
}

######## transcript DNA

convierte_adn = function(data)
{
largo_ADN = sum(TransMatrix)+1;
cadena1=rep(0,34);
cadena2=rep(0,102);
cadena3=rep(0,34);
tripleta=rep(0,3);
secuencia=rep(0,1);
###Number of cicles for exact match

###real length chain "cadena" = dim(data)[1]
exacta=0;
distancia=100;
for (i in 1:1)
	{
	contador=0;
	cadena1 = data[i,1:34];
	check1=data[i,35];
	check2=data[i,36];
		for(repeticion in 1:cicles)
		{
		cadena2=buscar(cadena1[1]);
		cadena3[1]=((2*(cadena2[1]*cadena2[3]))-
(3*cadena2[2]));

		for (j in 2:34)
			{
			tripleta = buscar(cadena1[j]);
			cadena2=c(cadena2,tripleta);

	cadena3[j]=((2*(tripleta[1]*tripleta[3]))-
(3*tripleta[2]));
			}
		###print(cadena1);
		###print(cadena2);
		###print(cadena3);
		suma1=0;
		suma2=0;
		for (k in 1:34)
			{

	suma1=ifelse(is.odd(k)==TRUE,suma1+cadena1[k]
,suma1-cadena1[k]);

	suma2=ifelse(is.odd(k)==TRUE,suma2+cadena3[k]
,suma2-cadena3[k]);
			}
		exacta = ifelse(check2==suma2,1,0);

		if(exacta==1)
			{
			elegida=cadena2;
			repeticion=cicles;
			exacta==0;
			##print(elegida);

	##print(c(repeticion,suma1,suma2));
			##print(distancia);
			distancia=0;
			}
		else
			{
			if((abs(check2-suma2))<10)
				{
				elegida=cadena2;
				distancia=abs(check2-
suma2);
				##print(elegida);

	##print(c(repeticion,suma1,suma2));
				##print(distancia);
				}
			}

			}
	secuencia=c(secuencia,elegida);
	}

return(secuencia);
}
########
recodifica= function(data)
{
text=substring(data[1], 1:17, 1:17)
text2=decodificada[c(2,3)];
DNA=c(text,text2);
i=4;
columnas=1;
while(i<=length(data))
	{
	text= substring(data[i], 1:17, 1:17)
	text2=decodificada[c(i+1,i+2)];
	text3=c(text,text2);
	DNA=rbind(DNA, text3);
	columnas=columnas+1;
```

```
        i=i+3;
        }
rownames(DNA)=c(1:columnas);
return(DNA);
}


codificador = function(sequency)
{
contador=0;
calibrador=rep(0,34);
sumatoria=rep(0,34);
codificacion=rep("0",21);
codificacion[18]="/";
codificacion[20]="=";

j=1;
while (j<=length(sequency))
        {
        contador=contador+1;
        sumatoria[contador]=strtoi(sequency[j])+strtoi(sequ
ency[j+1])-strtoi(sequency[j+2]);
        calibrador[contador]=strtoi(sequency[j])+strtoi(sequ
ency[j+1])+strtoi(sequency[j+2]);
        j=j+3;
        }

valor=0;
contador=0;
j=1;
while (j <=length(sumatoria))
        {
        valor=strtoi(paste0(sumatoria[j],sumatoria[j+1]))
        valor=valor+33;
        contador=contador+1;
        codificacion[contador]=chr(valor);
        j=j+2;
        }
valor=0;
valor2=0;
for (i in 1:length(sumatoria))
        {
        if(is.odd(i)==TRUE)
                {
                valor=valor+sumatoria[i];
                valor2=valor2+calibrador[i];
                }
        else
                {
                valor=valor-sumatoria[i];
                valor2=valor2-calibrador[i];
                }
        }

codificacion[19]=toString(valor);
codificacion[21]=toString(valor2);
return(codificacion);
}


###measures the time machine for process the code
trials=1
tiempo <- proc.time()

#Main Decoding Sequence
datos = scan("compacted.txt", what=character(1));

for (i in 1:16)
        {
        substring=word(datos[1],(i+1),sep="-")
        TransMatrix[i]=strtoi(substring);
        }
decodificada=lectura(datos);
probability = recodifica(decodificada);
numeros = numerifica(probability);
cicles=100000;
                accuracy=80;
            proc.time()-tiempo

                for (x in 1:trials)
                        {
        t2 <- proc.time() ##function that measures time
                ADN_seq = convierte_adn(numeros);
                ####ends time measuring sequence
                        proc.time()-t2
                tiempo=rbind(tiempo,t2);
                accuracy=accuracy+10;
                        cicles = x*100;
                tiempo[x+1,4]=cicles;
                tiempo[x+1,5]=accuracy;
                        }
```