Contents

Genetic Sequence Alignment

Optimal Substructure

Consider an optimal alignment of X,Y as X+spaces,Y+spaces such that they have the same length |X| = |Y|. Focus on the right most character. There are 3 relevant possibilities for the content of the final position for X,Y. Let's start with final position of X. Observe that if that's a character of the string X, it can only be the very last character x_n. That's because that's where this string ends. We don't know that little x_n is in the final position, there might be a space. Similarly, in the Y sequence of this final position, there's two possibilities. It has to be the final character, y_n. So that one seems to suggest 4 possibilities, two options for X sequence, two options for Y sequence. But we can see that it is totally pointless to have a gap in both sequences.

The penalty for gaps is non-negative, so if we just deleted both of those gaps we'd get an even better alignment of X and Y and in studying an optimal solution, we can therefore assume we never have two gaps in a common position. This assumption leaves us with 3 cases:

1. It could be that this alignment matches the character x_m with y_n.
2. It could match the final character of X, with a gap.
3. It could match the final character of Y with a gap.

This means that there are only 3 candidates. one candidate for each of the three possibilities for the contents of the final position. The smaller sub-problem is going to involve everything except the character in the final position. So it's going to involve X and Y, possibly with one character remaining. So let's let X' be X, with its final character peeled off. Y' be Y with its final character peeled off.

Assume case 1 holds. This means that the contents of the final position, includes both of the characters x_m and little y_n. So now what we're going to do is we want to look at a smaller sub problem. We want to look at the subproblem induced by the contents of all of the rest of the positions. We're going to call that the induced alignment. Since we started with an alignment, two things that had to equal length, and we peeled off the final position of both, we have another problem that has equal length so we're justified in calling it an alignment. In case 1, that means what's missing from the induced alignment are the final characters x_m,y_n which means the induced alignment is a bona fide alignment of X' and Y'. We are hoping that the induced alignment is an optimal alignment of these smaller X' and Y'. This would say that when we're in case 1, the optimal solution to the original problem is built up in a simple way from an optimal solution to a smaller subproblem. In case 2, the missing character from the induced alignment is the final character of X. So it's going to be the induced alignment of X' and Y. Similarly, in case 3, the induced alignment is going to be an alignment of X and Y'.

We're going to assume the contrary, we're going to assume that the induced solution to the smaller subproblem is not optimal, and from the fact that there is a better solution for the subproblem, we will extract a better solution for the original problem, contradicting the purported optimality of the solution that we started with. So when we're dealing with case 1, the induced alignment is of the strings X' and Y', X and Y with the final character peeled off. And so for the contradiction, let's assume that this induced alignment, it has some penalty, say P. Let's assume it's not actually an optimal alignment of X' and Y' i.e. suppose if we started from scratch we could come up with some superior alignment of X' and Y', with total penalty P*, strictly smaller than P. But if that were the case, it would be a simple matter to lift this purportedly better alignment of X' and Y' to an alignment of the original X and Y. We just reuse the exact same alignment of X' and Y', and in the final position, we just match x_m with y_n.

So the total penalty of this extended alignment of all of X and Y is the penalty incurred in everything but the final position, and that's just the P*+ new penalty incurred in the final position. And that's just the penalty corresponding to the match of the characters x_m and y_n. P* < P, of course P*+ alpha($x_m y_n$) < P + alpha($x_m y_n$). But this second term is simply the total penalty incurred by our original alignment of X and Y. But that furnishes the contradiction that we suppose that we started with an optimal alignment of X and Y, yet here is a better one, a contradiction proving optimal substructure.

Greedy Choice

$i \leftarrow 0$
while $i < \min\{M, N\}$ and $a_{i+1} = b_{i+1}$ do
    $i \leftarrow i + 1$
$R(0, 0) \leftarrow i$
$d \leftarrow L \leftarrow U \leftarrow 0$
repeat
    $d \leftarrow d + 1$
    $L \leftarrow L - 1$
    $U \leftarrow U + 1$
    for $k \leftarrow L$ to U do

$$i \leftarrow \max \begin{cases} R(d-1, k-1) & \text{if } L+1 < k \\ R(d-1, k) + 1 & \text{if } L < k < U \\ R(d-1, k+1) + 1 & \text{if } k < U-1 \end{cases}$$

        $j \leftarrow k - i$
        while $i < M$, $j < N$ and $a_{i+1} = b_{j+1}$ do
            $i \leftarrow i + 1$; $j \leftarrow j + 1$
        $R(d, k) \leftarrow i$
        if $i = M$ then $L \leftarrow k + 2$
        if $j = N$ then $U \leftarrow k - 2$
until $R(N - M, d) = M$
write "Number of differences is" d

Proof

Let the score have mat = match, mis = mismatch and ind = insertion/deletion. Assume that any two alignment $X_1..X_n$ and $Y_1..Y_n$ have the same score.

Pick any alignment that has >1 mismatch. One of these can be replaced by a deletion followed by insertion and by changing one of the entries, the mismatch can be a match. This methodology leaves the score unchanged such that $2 \times mis = 2 \times ind \times mat$. This implies that $ind = mis - \frac{mat}{2}$.

Suppose the scoring satisfy ind = mis – mat/2, then any alignment $X_1..X_n$ and $Y_1..Y_n$ with d. differences has score S'(i+j, d) = (i, j) * mat/2 - d * (mat - mis). Consider such an alignment. with J matches, K mismatches and I insertions/deletions, and let k = i+j. Then k = 2J – 2K + I, since each match or mismatch extends the alignment for two anti-diagonals, while each insertion or deletion extends it by one. For instance, a substitution from a vertex (u, v) in the grid extends the alignment for the two anti-diagonals u + v + 1 and u + v + 2. While an insertion or deletion from the same vertex extends it by the anti-diagonal u + v + 1 only.

Thus, alignment have d differences where $d = K - I$, while its similarity score is:

$$\text{matJ} + \text{misK} + \text{indI} = \text{mat}\frac{\text{k}-2\text{K}-\text{I}}{2} + \text{misK} + \text{I}\left(\text{mis} - \frac{\text{mat}}{2}\right) = \text{k}\frac{\text{mat}}{2} - \text{d}(\text{mat} - \text{mis})$$

It follows that if $ind = mis - \frac{mat}{2}$, then minimizing the number of differences in an alignment is equivalent to maximizing its score.

Dynamic Programming

```
for i=0 to length(A)
  F(i,0) ← d*i
for j=0 to length(B)
  F(0,j) ← d*j
for i=1 to length(A)
  for j=1 to length(B)
    F(i,j) = max { F(i-1,j-1) + s(A_i, B_j)
                 { F(i-1,j) + d
                 { F(i,j-1) + d
AlignmentA ← ""
AlignmentB ← ""
i ← length(A)
j ← length(B)
repeat until (i > 0 and j > 0)
  if (i > 0 and j > 0 and F(i,j) == F(i-1,j-1) + S(A_i, B_j))
    AlignmentA ← A_i + AlignmentA
    AlignmentB ← B_j + AlignmentB
    i ← i - 1
    j ← j - 1
  else if (i > 0 and F(i,j) == F(i-1,j) + d)
    AlignmentA ← A_i + AlignmentA
    AlignmentB ← "-" + AlignmentB
    i ← i - 1
  else
    AlignmentA ← "-" + AlignmentA
    AlignmentB ← B_j + AlignmentB
    j ← j - 1
```

Proof

Given two sequences a and b of lengths n_1 and n_2 respectively, an alignment A(a,b) is a sequence of pairs:

$$[(a_{i1}, b_{j1}), (a_{i2}b_{j2}), \ldots . : 1 \leq i_1 < i_2 < \cdots \leq n_1, 1 \leq j_1 < j_2 \ldots \leq n_2]$$

Given an alignment A(a,b), we let

$$M = \left|\{k: a_{ik} = b_{jk}\}\right|, X = |\{k: a_{ik} \neq b_{jk}\}|$$

and

$$S = |\{r: a_r \neq i_k \; for \; any \; k\}| + |\{r: b_r \neq j_k for \; any \; k\}|$$

In other words, M counts the number of matching characters in the alignment, X counts the number of mismatches and S the number of "spaces", i.e. characters not aligned to another character in the other sequence. Since every character must be either matched, mismatched, or a not aligned, we have that

$$2M + 2X + S = n_1 + n_2$$

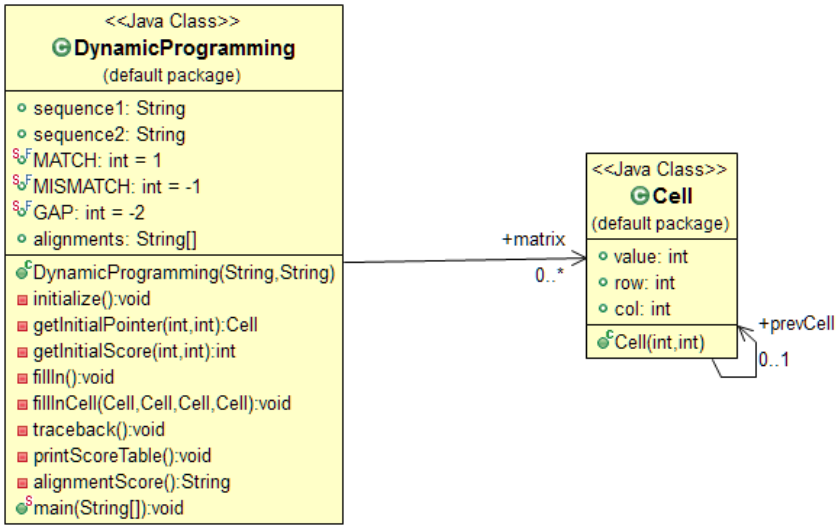
Given scores (parameters) consisting of real numbers m, x, s, the algorithm finds an alignment that maximizes the function $m \times M + x \times X + s \times S$. Note that by the linear relationship between M, X and S described above, there are really only two free parameters, and without loss of generality we can assume they are s and x. Furthermore, only S and X are relevant for computing the score of the alignment. The scores m, x and s can be interpreted as logarithms of probabilities if the sign on the parameters is the same, or as log-odds ratios if the signs are mixed.

Theoretical Analysis of Complexity

To find the asymptotic complexity of the algorithm, we need to analyse each individual part of the algorithm. To initialize the matrix, we need to input the scores of the row 0 and column 0 with d*j and d*i. This has a time complexity of Θ(M+N). Then we fill in the matrix with all the scores F(i,j) and mark it according to the rules. For each cell of the matrix, three neighbouring cells must be compared, which is a constant time operation. To fill the entire matrix, the time complexity is the number of entries, or Θ(MN). Finally, the traceback involves traversing the final path. This step can include a maximum of N cells (where we N>M), this step is Θ(N). Thus, the overall time complexity of the algorithm is $\Theta(\mathrm{M} + \mathrm{N}) + \Theta(\mathrm{MN}) + \Theta(\mathrm{N}) = \Theta(\mathrm{MN})$. The total space complexity of this algorithm is Θ(MN) since it fills a single matrix of size MN.

In comparison, two 12 residue sequences would require considering approximately 1 million alignments linearly while 150 residue sequences would require considering approximately 10^{88} alignments (more than the estimated number of atoms in the universe).

Implementation and Results



The basic idea is that the matrix is composed of a Cell objects. Cell objects also have a reference to their 'parent'. Because of this we have to fill every F(i,j) with a Cell object during initialisation and we need to store a 'parent' Cell so that trace back would we easier

FIGURE 1: CLASS DIAGRAM OF JAVA IMPLEMENTATION OF ALGORITHM

Reversed DNA		
Size of DNA	**TestID**	**Average time to complete (ms)**
3500	13	1975.533
3000	18	1317.333
2000	17	255.3333
1000	16	103.1333
100	1	1.933333

Mutated DNA (multiple mutations)		
Size of DNA	**TestID**	**Average time to complete (ms)**
3500	2	2364.2
3000	9	1360.467
2000	20	241.3333
1000	5	112.3333
100	14	0.2

Same DNA		
Size of DNA	**TestID**	**Average time to complete (ms)**
3500	15	2164.267
3000	12	1133.067
2000	3	338.0667
1000	19	94.06667
100	10	0.266667

Random DNA		
Size of DNA	**TestID**	**Average time to complete (ms)**
3500	4	664.3333
3000	6	291.8667
2000	7	107
1000	8	25.6
100	11	0.066667

The reversed DNA is the opposite of Sequence A such that $A_0 \ldots A_N = B_N \ldots B_0$. Both sequences are equal in length.

The mutated DNA is either an insertion, deletion or a replacement of a nucleotide in the original DNA i.e. Sequence A mutation. A replacement will result in a same-length sequence with a different nucleotide at a random position while an insertion will result in a longer Sequence B. Deletion will do otherwise.

As the name implies, same DNA is of same length which means a score of size S. On the other hand, a random DNA implies a randomly generated sequence of the specified size S. The size of Sequence B in a randomly generated DNA is S/2.

All the results are as expected. It follows that the complexity is $\Theta(MN)$.

Sample Output:

```
CAATGTGAATC
 GATCGGCAT
*-++--+-++* (-3)
```

Edit Distance Problem

Edit distance is a way of quantifying how dissimilar two strings are to one another by counting the minimum number of operations required to transform one string into the other.

Optimal Substructure

We are trying to minimize the number of transformations, so if we have the optimal solution to the three cases we consider (replace, insert and delete) then we get the minimum from those 3 and that's our optimal solution

Greedy Algorithm

Let
insertionCost be 1
deletionCost be 1
substitutionCost be 0 if $A_i = B_j$*, 2 otherwise*

if length(A) = 0
return length(B)
else if length(B) = 0
return length(A)

i ← length(A)
j ← length(B)

if $A_{i-1} = B_{j-1}$
return this $(A_{1..i-1}, B_{1..j-1})$
else

$$\textit{return } min \begin{cases} this\ (A_{1..i}, B_{1..j-1}) + insertionCost \\ this\ (A_{1..i-1}, B_{1..j}) + deletionCost \\ this\ (A_{1..i-1}, B_{1..j-1}) + substitutionCost(i-1, j-1) \end{cases}$$

Dynamic Programming

Let
insertionCost be 1
deletionCost be 1
substitutionCost be 0 if $A_i = B_j$*, 2 otherwise*

for i=0 to length(A)
F(i,0) ← i
for j=0 to length(B)
F(0,j) ← j
for i=1 to length(A)
for j=1 to length(B)

$$F(i,j) = min \begin{cases} this\ (A_{1..i}, B_{1..j-1}) + insertionCost \\ this\ (A_{1..i-1}, B_{1..j}) + deletionCost \\ this\ (A_{1..i-1}, B_{1..j-1}) + substitutionCost(i-1, j-1) \end{cases}$$

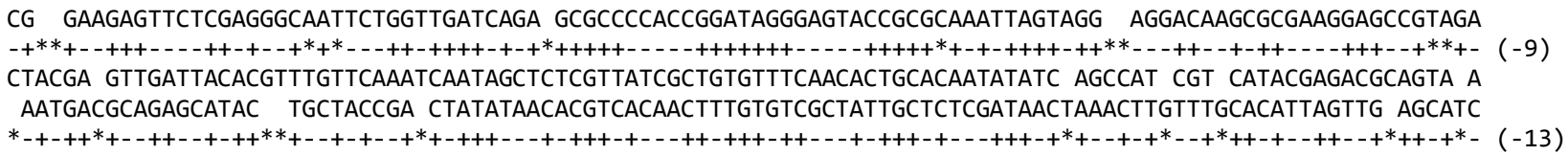
Edit distance is F(length(A), length(B))

Complexity

There are M * N entries which take constant time to compute therefore Θ(MN) space and time complexity

Appendix

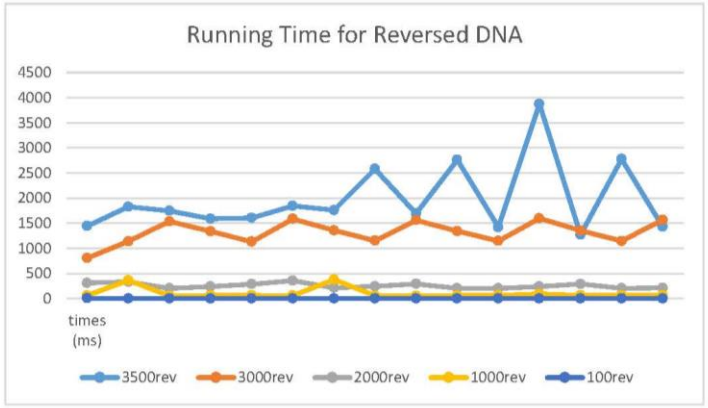
Sample Output

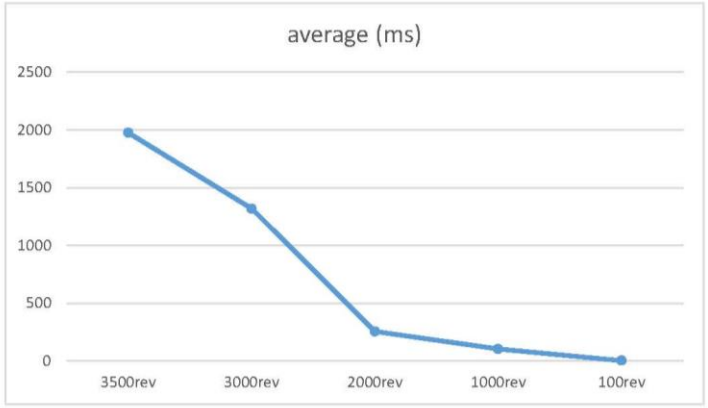

CG GAAGAGTTCTCGAGGGCAATTCTGGTTGATCAGA GCGCCCCACCGGATAGGGAGTACCGCGCAAATTAGTAGG AGGACAAGCGCGAAGGAGCCGTAGA (-9)
CTACGA GTTGATTACACGTTTGTTCAAATCAATAGCTCTCGTTATCGCTGTGTTTCAACACTGCACAATATATC AGCCAT CGT CATACGAGACGCAGTA A
AATGACGCAGAGCATAC TGCTACCGA CTATATAACACGTCACAACTTTGTGTCGCTATTGCTCTCGATAACTAAACTTGTTTGCACATTAGTTG AGCATC (-13)

Tables and Graphs

Reversed

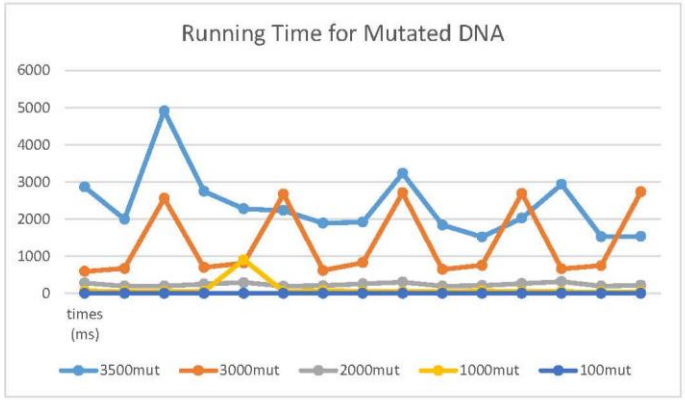
name	id	times (ms)															average (ms)
3500rev		13	1445	1827	1747	1589	1607	1846	1761	2580	1692	2764	1420	3877	1271	2779	1428	1975.533
3000rev		18	805	1140	1536	1338	1132	1585	1359	1152	1559	1344	1148	1597	1354	1145	1566	1317.333
2000rev		17	311	330	204	240	287	357	215	242	293	203	204	241	290	203	210	255.3333
1000rev		16	60	364	53	60	63	57	379	54	53	61	63	92	63	63	62	103.1333
100rev		1	6	4	4	2	2	2	1	1	1	1	1	1	1	1	1	1.933333

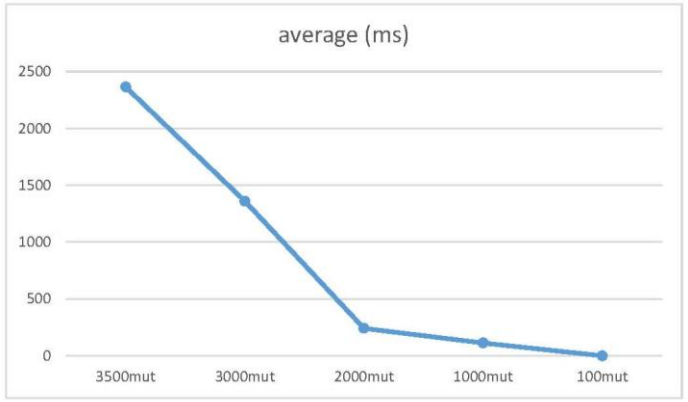




Mutated

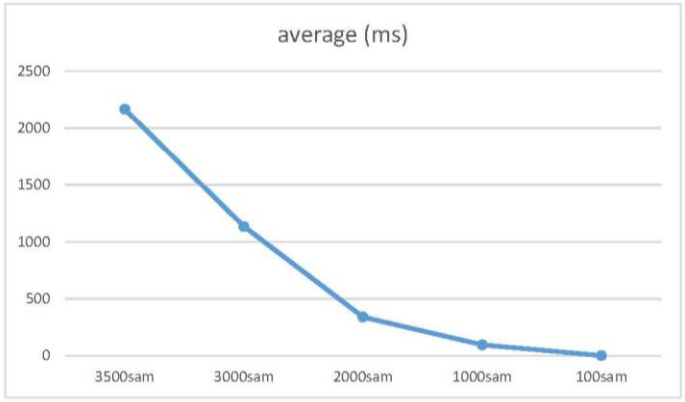
name	id	times (ms)															average (ms)
3500mut		2	2862	1998	4909	2750	2277	2234	1890	1923	3231	1842	1520	2025	2936	1530	1536	2364.2
3000mut		9	591	674	2557	699	818	2674	619	828	2709	646	756	2691	660	750	2735	1360.467
2000mut		20	281	199	200	252	293	196	211	260	305	200	215	266	318	201	223	241.3333
1000mut		5	66	54	53	55	908	54	75	50	51	51	72	51	47	49	49	112.3333
100mut		14	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0.2

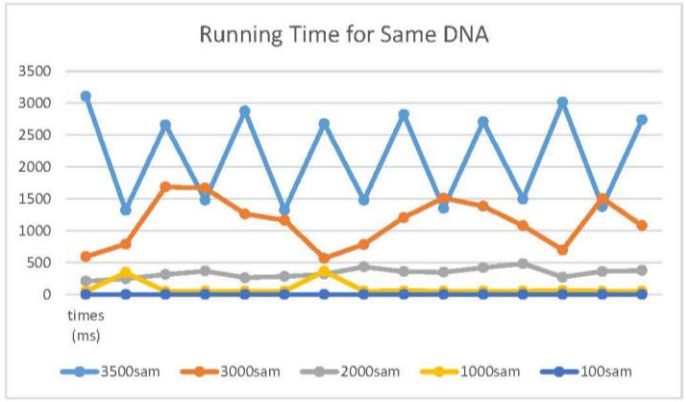




Same

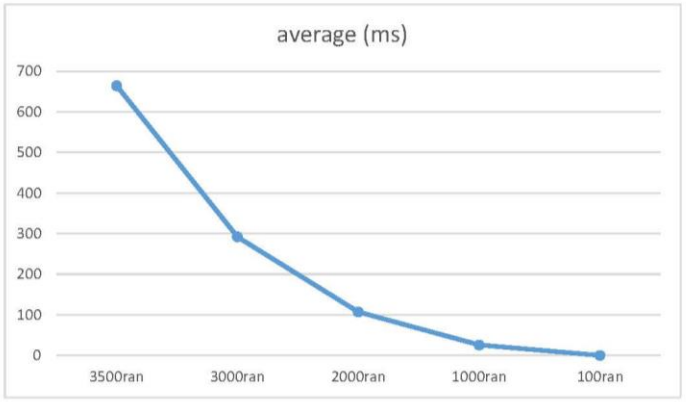
name	id	times (ms)															average (ms)
3500sam		15	3107	1322	2663	1480	2877	1323	2681	1480	2827	1353	2712	1499	3019	1380	2741	2164.267
3000sam		12	593	791	1687	1669	1264	1164	569	786	1207	1512	1385	1082	698	1508	1081	1133.067
2000sam		3	209	246	315	369	264	284	320	435	360	351	422	485	273	362	376	338.0667
1000sam		19	50	347	49	54	53	54	366	53	56	54	53	55	62	52	53	94.06667
100sam		10	0	1	0	1	0	0	0	0	0	0	1	0	0	1	0	0.266667

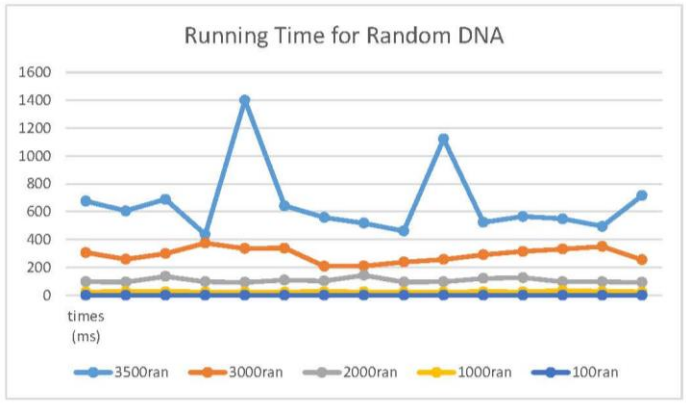




Random

name	id	times (ms)															average (ms)
3500ran		4	676	606	689	437	1401	643	559	518	462	1123	524	566	550	495	716	664.3333
3000ran		6	307	259	300	375	336	339	209	211	239	258	292	315	332	351	255	291.8667
2000ran		7	97	95	136	98	94	109	104	144	95	97	122	127	97	97	93	107
1000ran		8	24	27	26	24	24	24	26	25	24	23	26	25	34	28	24	25.6
100ran		11	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0.066667





Overall Running Times

name	id	times ms															average (ms)
3500mut	2	2862	1998	4909	2750	2277	2234	1890	1923	3231	1842	1520	2025	2936	1530	1536	2364.2
3500sam	15	3107	1322	2663	1480	2877	1323	2681	1480	2827	1353	2712	1499	3019	1380	2741	2164.267
3500rev	13	1445	1827	1747	1589	1607	1846	1761	2580	1692	2764	1420	3877	1271	2779	1428	1975.533
3000mut	9	591	674	2557	699	818	2674	619	828	2709	646	756	2691	660	750	2735	1360.467
3000rev	18	805	1140	1536	1338	1132	1585	1359	1152	1559	1344	1148	1597	1354	1145	1566	1317.333
3000sam	12	593	791	1687	1669	1264	1164	569	786	1207	1512	1385	1082	698	1508	1081	1133.067
3500ran	4	676	606	689	437	1401	643	559	518	462	1123	524	566	550	495	716	664.3333
2000sam	3	209	246	315	369	264	284	320	435	360	351	422	485	273	362	376	338.0667
3000ran	6	307	259	300	375	336	339	209	211	239	258	292	315	332	351	255	291.8667
2000rev	17	311	330	204	240	287	357	215	242	293	203	204	241	290	203	210	255.3333
2000mut	20	281	199	200	252	293	196	211	260	305	200	215	266	318	201	223	241.3333
1000mut	5	66	54	53	55	908	54	75	50	51	51	72	51	47	49	49	112.3333
2000ran	7	97	95	136	98	94	109	104	144	95	97	122	127	97	97	93	107
1000rev	16	60	364	53	60	63	57	379	54	53	61	63	92	63	63	62	103.1333
1000sam	19	50	347	49	54	53	54	366	53	56	54	53	55	62	52	53	94.06667
1000ran	8	24	27	26	24	24	24	26	25	24	23	26	25	34	28	24	25.6
100rev	1	6	4	4	2	2	2	1	1	1	1	1	1	1	1	1	1.933333
100sam	10	0	1	0	1	0	0	0	0	0	0	1	0	0	1	0	0.266667
100mut	14	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0.2
100ran	11	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0.066667

Overall Average