Question 1:

Result of Rabin Karp Algorithm

Text: accacatcaaatgaaaatgggtccattgattgccccttcaattcactacttcagcctcactcgagaggcactattgttcaggcttaggaccgcgagacgatcgggtccagggggctgtggcacgcctctcgcaatgaa….of length 1000

Pattern: agttactttggaaaagggtgact

Running Time Table:

|  |  |
| --- | --- |
| Modulus Size | Running Time (in ms) |
| 2, 3, 5, 7 | 22 |
| 1013 | 20 |
| 5003 | 18 |
| 7019 | 15 |
| 10007 | 15 |
| 100019 | 15 |
| 1000037 | 15 |

From the above table we can notice that, increasing the size of the modulus decreases the running time.

This is because when the modulus increases, the number of hash buckets increase and thus the probability of collision decreases. Since, we do a brute force comparison only when the collision occurs, time spent in brute force comparison is lesser when the modulus is greater.

A modulus of 2 gives the result in 22 ms and a modulus of 1000037 gives the result in 15 ms.

The smallest modulus from where the running time starts stabilizing to a minimum value is approximately 7001.

Question2:

The Smith Waterman algorithm has been run on human and chimpanzee DNA sequences (of length 600 each)

The following is the output for the same:

Using standard insertion/deletion/substitution cost value:

alignment = MDMIMMMSSSMMMSMMMSMMSSMMMIMMSMSMMMDMSMMMDMISMMSMMDDSMMMMDMDMMDMMISSSMMMMMDDMISSMMIMMMDDSSMMMDSMMIIMMDDMIMMIIMSMSMMMMDMDSMSSMMIMDMMMMDMMDDMMDDMSMSSMDMIIMMDSSMMDDMMSMMMSMDMSMDMMDMSMMMSMMSMISMDSMMSSMISMIMISSMIMMIMMMIMIMMMMISMMDDMMMMISMMMMIMDMMMMIMDSMMDDSMMSMSMMMIMSMSSSMDMSSMSMMSMMIMMSMMMIMDMIIMIMIMSMMSMMMISSMMMDMSMSMMDDMSMDMMSMMDMSMMMMIMMMIISMMMMDSMMMMMIIMIIIIMMIISMIIMMMIIMSSMMISMMIMMIIIISMIMMIMIIMIMMMMMSMMIIMMSMMIMIMSMMDMMMIISMMIIISMMMIIMSSMMDSMMISMMDMMMMSSMMMMMSMDMMMIMMDSSMDMSSMMIMMMIMSMMIMMMMDDSMDMMMDMDMMDMMMSMMMSMMIIMMMIMIIIMMSMIIMMIMMDSMSMMIMMSSMDSSMMMMIIMM

alignment score = 584

Using Blosum62 score matrix:  
alignment = DSMMSSSMSSMMMSDSSMMMIMMDMSSMMISMMMIIISMMIMMIMSMIMSSMSSMSSMMSMSSSMSSSMMMMSMMSMSMMDSSMSSSSSMSMSMSSSMMSMSMDDMMMSISMSMSSMMSDMMDMSSMSMMMSIMMMDMSMSSDSMDMMDMMMMMMIMMSSSSSSMSSSMSMMDDSSMMMSMMMMMMDDDMMDMSMMSMDMDMMSMMSSIMMIMIIMMIMIMMIIMIMMMMSSDDMSSMISMMMIMMMSIMDMDMSMMDSMMMMMMSSMDMSMSMIMIIMMMMMMDMMDDMSSMMMSMSSMMIMISMSSMSMMMSMSSSMMDMDMMMIMMMIMIIISMSMMMISMDSSSMSSMDDSSMMSSSMSMMSSMDDMMDMMDMSMMMIMMDMSMMSMMDMISMSMMMSMMMSSMMDDMMSSSMMMMMMSMISMIIIMMIMMSMMMSSSMSSSSMMMSMIMIMMDMMSSSMMIMSMSMIMMMSSSMMIMIMSSMMMSMIMMMMSSMDMMSMISSSSMSMSMSSMDMMSSMMIMIMSMMISMMMSMSSMMMIMSMSMMIIMIMSMMISSMMMMMMSDMDMMISMMMSMMSSMSMSISMSMSMMMMDSSSMSSMMSSSMMMMMSMDMMSMMIISSSMSMMSMIMSSMDMDSMSIMSMSMMISMSSMMSMSSSMSMSSMMDMMMSMSMDDSMMMMMDMSMDMSMMSMSSSSMISMDDMSMSSMMSMMMDMMDMDMDSMSSSMDSMSMMDMSMMSSMMSSM

alignment score = 1634

Using Blosum80 score matrix:

alignment = 

alignment score = 1255

The underlined areas show the areas of similarity in the DNA sequences of the human and the chimpanzee.

We see that there are significant number of local areas of similarity in the DNA sequences of the human and chimpanzee.

Thus, we can conclude that these two species have some features in common.

Question 3:

The two English articels taken are:

<http://www.footballfancast.com/football-news/manchester-united-1-0-arsenal-match-review>

<http://www.espnstar.com/football/premier-league/news/detail/item883954/Man-United-dispatch-Arsenal-with-ease/>

Both the articles describe the same event.

Results of Smith Waterman algorithm using Blosum62 score matrix:

alignment = MMMMMDIMMMMMMDIMMMMMMMMMMMDIMMMMMMMDDIIMMDDIIMMMMMMMMMMMMMMMMDIMMMMMMMMMMMMMDIMMMMMMMMMMMMMMMMDIMMMMMMMDIMMMMMMMMMMMMMMMMMMMMMMMMMMMDIMMDIMMMMMDIMMMMMMDIMMMMMMMMMDIMMMMDDIIMMMMDIMMMMMMDDIIMMMMMMMMMDIMMMMMDIMMDDIIMMMMMMMMMMDIMMMMMMMMDIMMDIMMMMMMMMMMMMMMMMMMMMMDIMMMMMMMMMMMMDIMMMMMMMMMMMDIMMDDIIMMMMMMMMMMMMMMMMDIMMDIMMMMMMMMMMMMMMDIMMMMMMMMMMDIMMMMMMMMMMDDIIMMDIMMMMMDIMMDIMMMMDIMDIMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMDIMMMMMMMMMDIMMMMMMMMMMMMMDIMMMMMDIMMMMMMMMMMMDDDIIIMDIMMMMMDDIIMMMMMDDIIMMDDIIMMMMMMMMMMDIMMMMDIMMMMMMDIMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMDDIIMMMDDIIMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMDIMMMMMMMMDIMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMDIMMDIMMMMMMMMMMMMMMMMMMMMMMMMMMMMDIMMMMMDIMMMMMMMMMMMMMMMMMMMMMMMDDIIMMMMMMMMMMDIMMMDDIIMMMDIMMMMMMMDIMMMMMMMDDIIMMMMMMMMMDIMMMMMMMMM

alignment score = 3478

local areas of similarity:

The underlined areas show the areas of similarity in the two news articles.

We see there that are significant number of local areas of similarity in the two news articles.

Thus, we can conclude that these two articles have a lot in common in terms of the idea, agenda description words and the people and the organizations involved.

The first article in the above program is translated to Spanish using Google translate and then SmithWaterman algorithm is run on them with Blosum62 score matrix.

The following is the output:

alignment = SSMDDSMSISSDISSISSMMSSSISISSMSISSSDIMDDSISDDISSSDMSMISSSISSSDMSDMSSSSSMMSIMSIMISDISSSIMDSISMMMIMMMSSSSSMSSMISDDSDIMSISSISSMMMMMMMMMMMMMMMMMMMDDDDDDIMDDDIMMDSISMDIMMMISSMSSIMDSDSDDDSSSSDIIMMMMMMDDIIMMMSSSSSMIIISIMMDSISDIMSDISMMDISMDDMDDDSMSSSISMSSISSDSIMMMMMMMMMMMMIISSIISMIIIMIIMMDDSSMSSIMMMMMMMDMISMISSMDDMDDDSDDDSMSSISDSSMSSISSMISDDDDDIMMMMMMMMMMMISIMSDSMMMMMMMMMDIMMMSDSISSISSDDSSDSDSISMDIMMDDDISSDDDISMSSDDSMSIMDDMDMMDDDDMSIMSSDIISSSSMSDSDMSSDSSSSMSSSIMDISISSMIMDMMMISMISSSMSSISMDIMMMMMISSIIIISMMSDDDSISMSISDSDDIISSSSSDDSSDDSSMSMIISMSMSDSMMSDIIIIMSSSMSISSSMSMSIIISMMSSMSDMMDDDMSSMMMSSSMDSMSSDDSSSDIISSISIISMMSSIIMMMIISMSISSSSMISSMMISDSISSSSSMIMSISSISISSSDSSSDSSSISSDIMSSSMSSSSIMISSM

alignment score = 777

The underlined areas show the areas of similarity in the two news articles.

We see there that are significant number of local areas of similarity in the two news articles.

Thus, we can conclude that these two articles have a lot in common in terms of the idea, agenda description words and the people and the organizations involved.

Question 4:

ORF or Open Reading Frames are those regions which do not have any stop codons. They begin with the start codon and end with either of the 3 stop codons.

ORF Distribution Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Length of ORF | Virus(25 ORF’s) | Bacteria(29 ORF’s) | Human(13 ORF’s) | Random(18 ORF’s) |
| 50-70 | 8% | 3.44% | 7.7% | 5.88% |
| 70-90 | 0% | 6.89% | 0% | 0% |
| 90-110 | 0% | 0% | 15.39% | 0% |
| 110-130 | 0% | 0% | 7.7% | 0% |

Every input genome sequence taken was more than 1000 base pairs. The threshold of the distribution table has been set lower to 50 as length of ORF’s can change corresponding to different gene sequences and whether they contain longer ORF’s or not.

From the above table,

We can notice that even though, the number of ORF regions found in the lower hierarchy organisms is more, longer sequence ORF’s are found in higher organisms such as Humans. Random sequence generated 18 ORF’s with lesser distribution than bacteria.

Longer ORF’s are identified as they are used to identify the protein coding regions in the DNA. Humans have the highest ORF regions allowing us to identify coding regions more prominently than any other organism.

Output:

ORF Lengths for the virus sequence:

7 39 7 2 16 7 1 29 12 35 14 9 1 1 67 58 25 10 20 7 24 12 42 15 41 1 15

ORF Lengths for the baceria sequence:

3 4 11 2 10 0 22 34 22 8 1 2 7 12 44 13 75 58 23 10 13 13 10 29 5 43 4 26 2 9 73

ORF Lengths for the human sequence:

23 116 107 8 28 8 7 31 42 37 17 15 108

ORF Lengths for the random sequence:

14 19 19 2 6 49 54 38 9 14 15 0 4 8 5 7 0 7

Question 5:

(a)Cchocohilaptes : It is a shuffle of chocolate and chips as on traversing from left to right we can traverse every alphabet of the word in the desired order.

Chocochilatspe: It is not a shuffle of the words chocolate and chips as on traversing from left to right, in case of chips, the alphabet ’s’ comes before ‘p’ violating the left to right rule.

(b)We can construct a matrix of row size equal to length of X string and column size equal to length of Y string. We initialize the first row and column of the matrix true wherever there is a match in between the string X and string Z and string Y and String Z. Then we compare for every position in the matrix that the first i characters of X and first j characters of Y, should be found in i+j characters of Z.

Boolean String\_Shuffle(String X, String Y,String Z) {

int x\_len = x.length();

int y\_len = y.length();

int z\_len = z.length();

if(z\_len!=x\_len+y\_len)

return false;

bool matrix[x\_len][y\_len];

matrix[0][0] = true;

for (int i = 1; i <x\_len; i++)

if (z[i-1] == x[i-1])

matrix [i][0] = matrix[i-1][0]&& TRUE;

else

matrix [i][0] = FALSE;

for (int j = 1; j < y\_len; k++)

if (z[j-1] == y[j-1])

matrix [0][j] = matrix[0][j-1]&& TRUE;

else

matrix [0][j] = FALSE;

for (int i = 1; i < n; i++)

for (int j = 1; j < m; j++) {

matrix[i][j] = (if (z[i+j-1] == x[i-1]) && matrix[i-1][j] )|| ( if(z[i+j-1] == y[j-1]) && matrix[i][j-1]);

}

return matrix [n-1][m-1];

}

Question 6:

The algorithm to make a palindrome from the given String with minimum number of insertions

Let the input string be str.

We use Dynamic Programming to solve this problem. We construct two matrices of size n x n (n being the size of str). They are the cost matrix and the parent matrix.

The values in the cost matrix are defined as follows:

m[i][j].cost = m[i+1][j+1].cost if str[i]==str[j]

= min(m[i][j+1].cost, m[i+1][j].cost) + 1 if str[i] != str[j]

This corresponds to the operation as follows:

i starting from the left end and j starting from the right end. We compare str[i] and str[j]. If they are equal, we proceed with further values of i and j. If they are unequal, we insert str[i] to the right of str[j] or insert str[j] to the left of I (whichever yields the minimum number of insertions).

We define three operations to be filled in the parent matrix:

N = no operation

L = insert str[j] to the left of str[i]

R = insert str[i] to the right of str[j]

The values in the parent matrix are defined as follows:

m[i][j].parent = N if str[i]==str[j]

else

= R if min(m[i][j+1].cost, m[i+1][j].cost) = m[i+1][j].cost

= L if min(m[i][j+1].cost, m[i+1][j].cost) = m[i][j+1].cost

To construct the result string i.e. the palindrome with minimum number of instructions, we traverse the parent matrix and reconstruct the path and then perform the sequence of operations on the input string.

int string\_compare(String s)

{

int i,j,k; /\* counters \*/

int opt[3]; /\* cost of the three options \*/

for (i=0; i<s.length(); i++) {

row\_init(i,s);

}

String t = s.reverse();

for (i=1; i<strlen(s); i++)

for (j=1; j<strlen(t); j++) {

if(s[i] == t[j])

{

opt[N] = m[i+1][j+1].cost;

opt[L] = MAX;

opt[R] = MAX;

}

else

{

opt[N] = MAX;

opt[L] = m[i][j+1].cost + 1;

opt[R] = m[i+1][j].cost + 1;

}

m[i][j].cost = opt[1];

m[i][j].parent = N;

for (k=0; k<2; k++)

if (opt[k] < m[i][j].cost) {

m[i][j].cost = opt[k];

m[i][j].parent = k;

}

}

operationSequence = reconstruct\_path(s, 0, 0)

return operate(s, operationSequence);

}

row\_init(int i, String s)

{

for(int k=0; k<i; k++) {

m[i][str.length()-1-k].cost = 0;

m[i][str.length()-1-k].parent = -1;

}

}

reconstruct\_path(String s, int i, int j)

{

if (m[i][j].parent == -1) return;

if (m[i][j].parent == “N”) {

return “N” + reconstruct\_path(s,i+1,j+1);

}

if (m[i][j].parent == “L”) {

return “L” + reconstruct\_path(s,i,j+1);

}

if (m[i][j].parent == “R”) {

return “R” + reconstruct\_path(s,i+1,j);

}

}

operate(String s, String sequence)

{

i = 0;

j = s.length() - 1

for (int k=0; k < sequence.length(); k++)

{

if(sequence[k] == “L”) {

insert(result, s[j], i-1)

}

if(sequence[k] == “R”)

{

insert(result, str[i], j+1)

}

i++, j--

}

return result;

}

The algorithm has the time complexity if O(n^2).

Question 7:  
Let the input be str

Let the multiplication table be represented by m.

We try to find whether the input can be reduced to two ordered symbols whose product is the expected symbol a.

The two ordered symbols can be: the first character of the input and the one determined from the remaining input………..(1)

OR

The one determined from the remaining input and the last character of the input………..(2)

We use Dynamic Programming to solve this problem. We construct two matrices of size n x n (n being the size of str). They are the cost matrix and the parent matrix.

The values in the cost matrix represent the list of all possible products till that point.

The values in the cost matrix are defined as follows:

m[i][j].cost = list(str[i] \* m[i+1][j].cost , m[i][j+1].cost \* str[j])

This corresponds to the operation as follows:

i starting from the left end and j starting from the right end. We evaluate the product of the substring starting from i+1 and multiply it with compare str[i]. Also, we evaluate the portion of the String from 0 to j-1 and multiply it with str[j]. We select the operands that give a product of a and proceed.

We define three operations to be filled in the parent matrix:

N = return FALSE. The product of a is not possible in the given String.

L = take str[j] as the right operand and evaluate the left operand

R = take str[i] as the left operand and evaluate the right operand

The values in the parent matrix are defined as follows:

m[i][j].parent = N if m[i+1][j].cost != a and m[i][j+1].cost != a

else

= R if str[i]\*m[i+1][j].cost == a

= L if m[i][j+1].cost\* str[j] == a

To construct the result string showing the parenthesis, we traverse the parent matrix and reconstruct the path and then perform the sequence of operations on the input string.

int string\_compare(String s)

{

int i,j,k; /\* counters \*/

String t = s.reverse();

int opt[3]; /\* cost of the three options \*/

for (i=0; i<s.length(); i++) {

for (j=0; i<t.length(); j++)

cell\_init(i,j,s,t);

}

for (i=1; i<strlen(s); i++)

for (j=1; j<strlen(t); j++) {

m[i][j].cost = list (s[i]\*(m[i+1][j].cost);

}

return (a in (m[0][0].cost));

}

cell\_init(int i, int j, String s, String t)

{

if (i+j == s.length()-1) {

m[i][j].cost = s[i]

m[i][j].parent = -1

}

if (i-j == 1) {

m[i][j].cost = s[i] \* t[j]

m[i][j].parent = -1

}

}

We build a n\*n cost matrix where,

cost of m[i][j] represents all possible products that can happen within indices i,j.

For eg.

m[0,5] will indicate all possible multiplication answers with possible combination of parenthesis.

So,

m[0][5] = list< m[0][0]\*m[1][5], m[0,1] \*m[2,5], m[0,2]\* m[3,5], m[0,3]\*m[4,5], m[0,4]\*m[5,5]>

m[0,5] is our goal cell. And if we get **‘a’** in this goal cell we say there exists one possible combination of parenthesis which can give us answer **‘a’.**

int k = 0;

for(int i = 0; I <strlen(s); I ++) {

for(j = i ; j< strlen(s) - i; j++) {

m[j][j+k] .cost = list(m[j][j].cost \* m[j+1][j+k].cost, m[j][j+1].cost \* m[j+2][j+k].cost,…..,

m[j][j+k-1].cost \* m[j+k][j+k].cost)

}

k++;

}

Complexity:

m[0][strlen(s)-1] denotes goal cell. If we have a in the list of costs that this cell contains we can say we can get a with at least one possible combination of paranthesis.

The algorithm has the time complexity if O(n^2).