November 14, 2019

Report on Assignment -3 (DNA Sequence Alignment)

Submitted by

Suresh Kumar Amalapuram

Roll No. CS19RESCH11001

AdvancedDataStructures & Algorithms

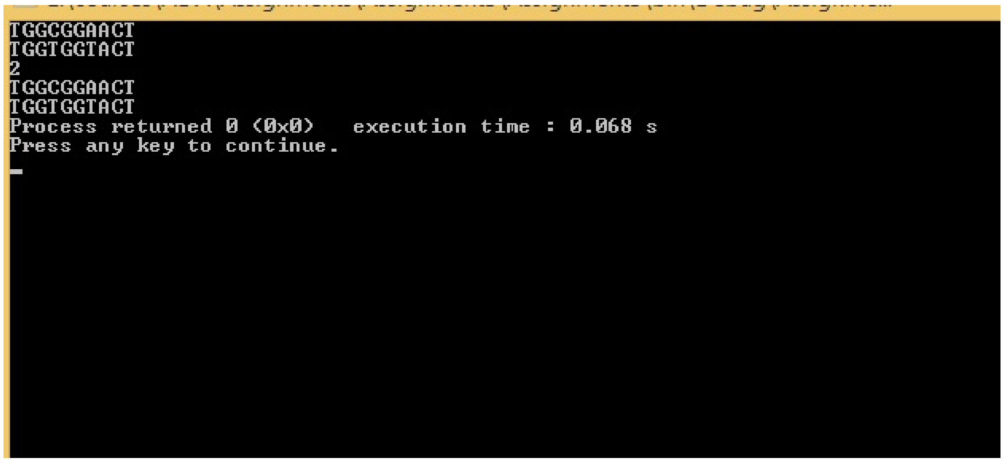
Chapter 1

Problem 1

Given two sequences, write C++ code to compute the cost, and alignment pattern of the optimal alignment.

1.1 Output Format

- compile the code and run executable file(in case of windows)/a.out file(in case of unix)
- Enter target string(source string/first string) in terminal window and hit ENTER
- Now you are in new line, enter query string (second string) hit enter
- You will see output in the following format
  - optimal cost value
  - new target string
  - new query string



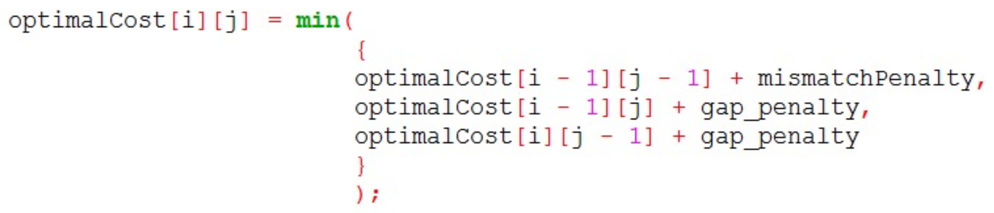
```
TGGCGGAACT
TGGTGGTACT
2
TGGCGGAACT
TGGTGGTACT
Process returned 0 (0x0)   execution time : 0.068 s
Press any key to continue.
```

Figure 1.1: Output Format

1.2 Solution/Strategy

1. we employ **Dynamic Programming paradigm** to solve this problem

2. The corresponding recurrence relations is given by



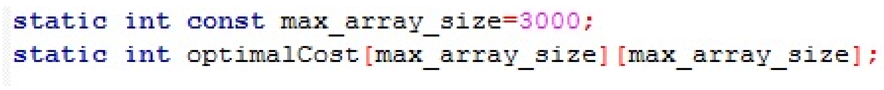
```
optimalCost[i][j] = min(
                        {
                        optimalCost[i - 1][j - 1] + mismatchPenalty,
                        optimalCost[i - 1][j] + gap_penalty,
                        optimalCost[i][j - 1] + gap_penalty
                        }
                        );
```

Figure 1.2: Recurrence for solution

3. A global Optimal Cost array was created, with maximum size 3000 x 3000 and a **global** variable **INFINITY** with a value of **25000** is used

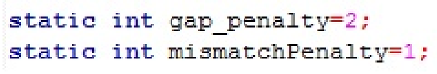
4. Code can handle any DNA string with length at most **3000**

5. The Corresponding penalties per the problem description used in the code



```
static int const max_array_size=3000;
static int optimalCost[max_array_size][max_array_size];
```

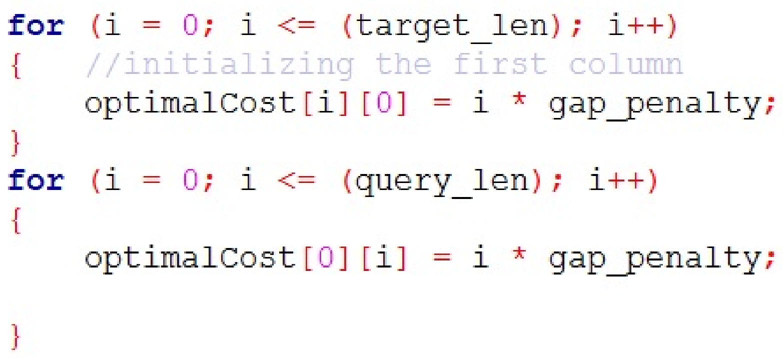
Figure 1.3: Optimal Cost Matrix



```
static int gap_penalty=2;
static int mismatchPenalty=1;
```

Figure 1.4: Penalties

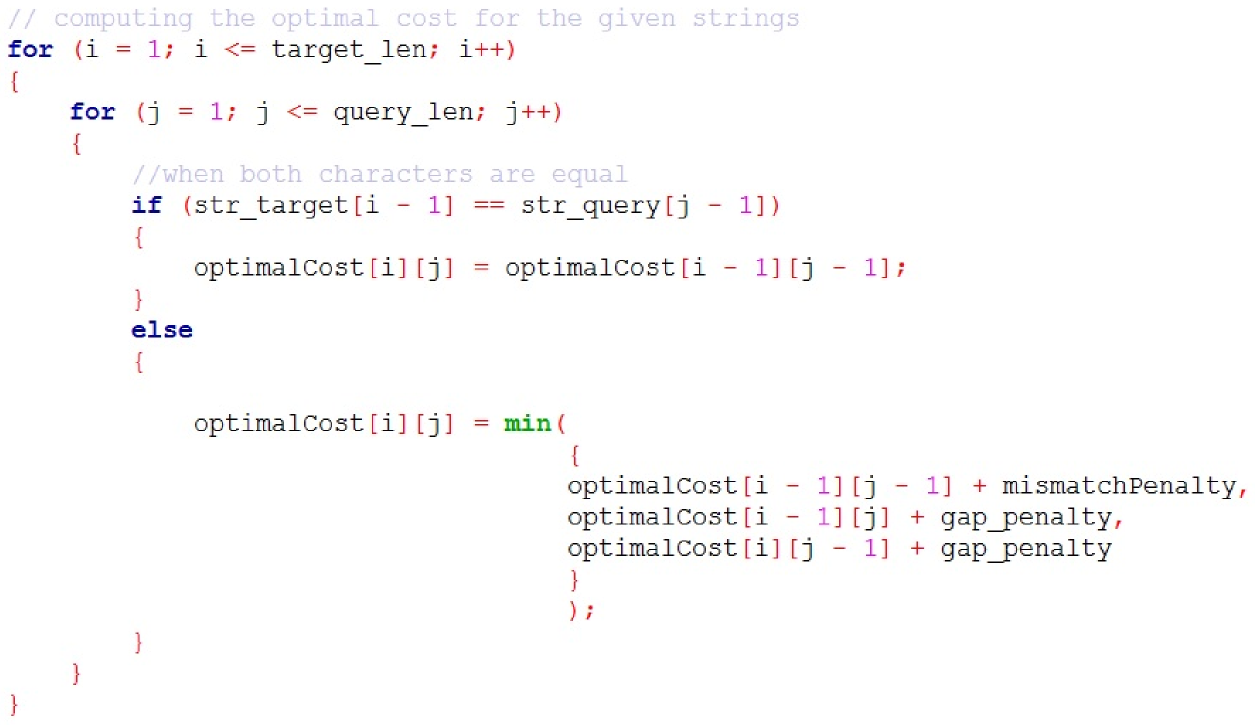
6. Optimal cost Matrix was initialized as follow



```
for (i = 0; i <= (target_len); i++)
{   //initializing the first column
    optimalCost[i][0] = i * gap_penalty;
}
for (i = 0; i <= (query_len); i++)
{
    optimalCost[0][i] = i * gap_penalty;

}
```

Figure 1.5: Optimal Cost Matrix initialization

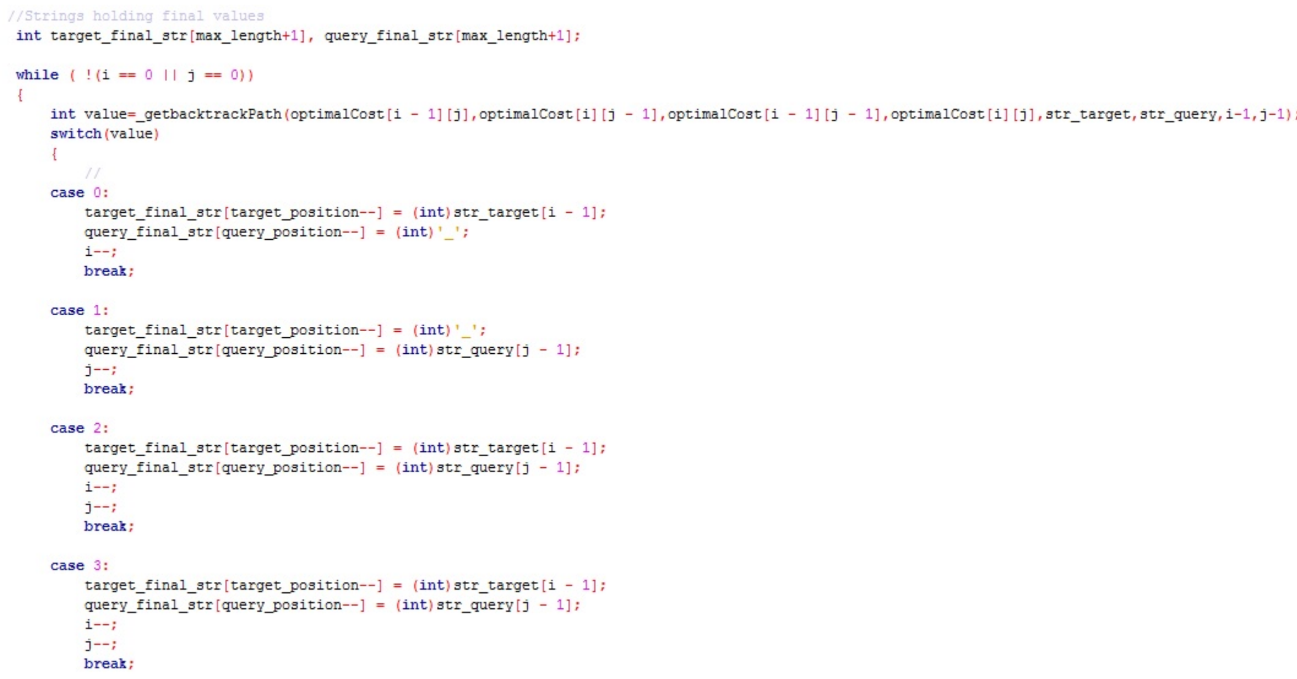
7. Optimal cost matrix is filled using recurrences as follow



```
// computing the optimal cost for the given strings
for (i = 1; i <= target_len; i++)
{
    for (j = 1; j <= query_len; j++)
    {
        //when both characters are equal
        if (str_target[i - 1] == str_query[j - 1])
        {
            optimalCost[i][j] = optimalCost[i - 1][j - 1];
        }
        else
        {

            optimalCost[i][j] = min(
                                    {
                                    optimalCost[i - 1][j - 1] + mismatchPenalty,
                                    optimalCost[i - 1][j] + gap_penalty,
                                    optimalCost[i][j - 1] + gap_penalty
                                    }
                                    );
        }
    }
}
```

Figure 1.6: Filling Optimal Cost Matrix

8. Once the Optimal cost matrix is filled, we use **back tracking** to print the output



```
//Strings holding final values
 int target_final_str[max_length+1], query_final_str[max_length+1];

 while ( !(i == 0 || j == 0))
 {
     int value=_getbacktrackPath(optimalCost[i - 1][j],optimalCost[i][j - 1],optimalCost[i - 1][j - 1],optimalCost[i][j],str_target,str_query,i-1,j-1);
     switch(value)
     {
         //
     case 0:
         target_final_str[target_position--] = (int)str_target[i - 1];
         query_final_str[query_position--] = (int)'_';
         i--;
         break;

     case 1:
         target_final_str[target_position--] = (int)'_';
         query_final_str[query_position--] = (int)str_query[j - 1];
         j--;
         break;

     case 2:
         target_final_str[target_position--] = (int)str_target[i - 1];
         query_final_str[query_position--] = (int)str_query[j - 1];
         i--;
         j--;
         break;

     case 3:
         target_final_str[target_position--] = (int)str_target[i - 1];
         query_final_str[query_position--] = (int)str_query[j - 1];
         i--;
         j--;
         break;
```

Figure 1.7: Back Tracking

1.3 Correctness of the Algorithm

Dynamic Programming uses recurrences, constitutes optimal solutions. so correctness focuses on proving recurrence relations are **correct**. This part is already covered in class so skipping it.

1.4 Efficiency of the Algorithm

1.4.1 Space Complexity

- Size of the Optimal Cost array = $O(mn)$
  - n = length of the Target DNA sequence
  - m = length of the Query DNA sequence
- Additional Space for storing DNA sequences = $O(m+n)$
- Memory requirements for auxiliary variables = $O(c)$

    - c=constant space

- Total Space Complexity $= O(mn+m+n+c) = O(mn)$

1.4.2 Time Complexity

- Time for Initialization(first column,first row) of the **Optimal Cost** matrix is $O(m+n)$
- Computation for each cell of the optimal cost matrix is 3 cases (constant)
- Total number of cells=$O(mn)$
    - n = length of the Target DNA sequence
    - m = length of the Query DNA sequence
- Time for back tracking =$O(m+n)$
- Total time Complexity $= O(mn+m+n+c) = O(mn)$

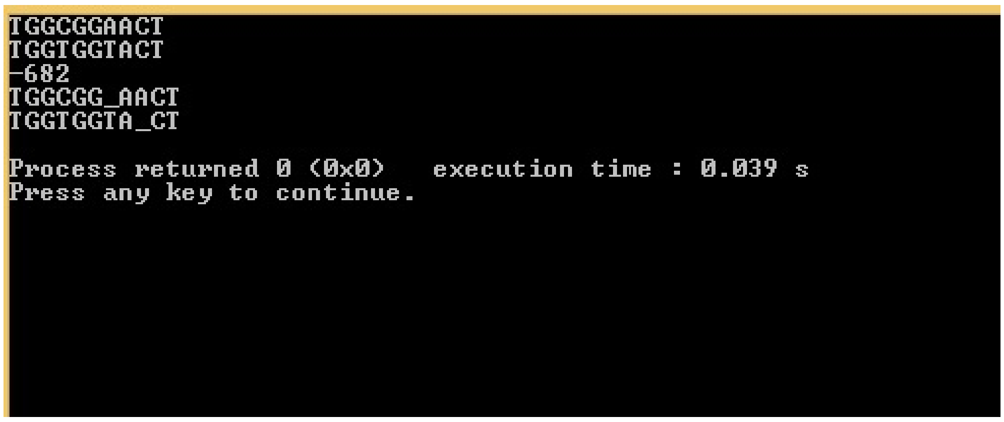
Chapter 2

Problem 2

Suppose that the matching penalties are specified by the following matrix (actually used in DNA sequence alignment),and a gap-penalty of $\gamma = 30$ applies for matching against the gap character.

2.1 Output Format

- compile the code and run executable file(in case of windows)/a.out file(in case of unix)
- Enter target string(source string/first string) in terminal window and hit ENTER
- Now you are in new line, enter query string (second string) hit enter
- You will see output in the following format
  - optimal cost value
  - new target string
  - new query string

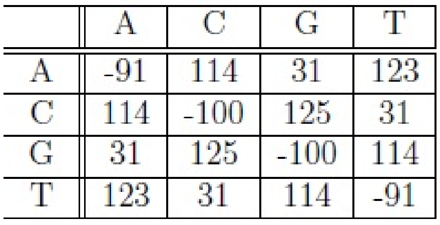


```
TGGCGGAACT
TGGTGGTACT
-682
TGGCGG_AACT
TGGTGGTA_CT

Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.
```

Figure 2.1: Output Format

2.2 Solution/Strategy

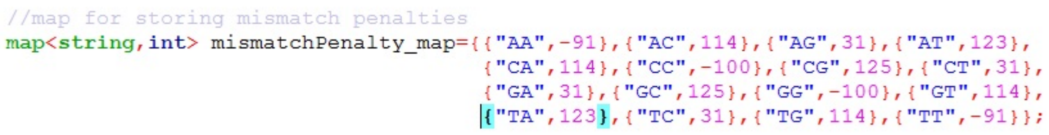
∗ Solution is similar to problem1, additionally we used the following Mismatch penalty matrix



	A	C	G	T
A	-91	114	31	123
C	114	-100	125	31
G	31	125	-100	114
T	123	31	114	-91

Figure 2.2: Mismatch Penalty Matrix

∗ Equivalent matrix has been encoded using **Map** from c++ STL as follow



```
//map for storing mismatch penalties
map<string,int> mismatchPenalty_map={{"AA",-91},{"AC",114},{"AG",31},{"AT",123},
                                     {"CA",114},{"CC",-100},{"CG",125},{"CT",31},
                                     {"GA",31},{"GC",125},{"GG",-100},{"GT",114},
                                     {"TA",123},{"TC",31},{"TG",114},{"TT",-91}};
```

Figure 2.3: Mismatch Penalty Map

2.3 Correctness of the Algorithm

Dynamic Programming uses recurrences, constitutes optimal solutions. so correctness focuses on proving recurrence relations are **correct**. This part is already covered in class so skipping it.

2.4 Efficiency of the Algorithm

2.4.1 Space Complexity

$*$ Size of the Optimal Cost array = $O(mn)$

- n = length of the Target DNA sequence
- m = length of the Query DNA sequence

$*$ Additional Space for storing DNA sequences = $O(m+n)$

$*$ Memory requirements for auxiliary variables = $O(c)$

- c=constant space

2.4.2 Time Complexity

- Computation for each cell of the optimal cost matrix is 3 cases (constant)
- Total number of cells=$O(mn)$
- n = length of the Target DNA sequence
- m = length of the Query DNA sequence
- Time for back tracking =$O(m+n)$
- Total Space Complexity $= O(mn+m+n+c) = O(mn)$

Chapter 3

Problem 3

Given two sequences, write C++ code find the optimal alignment (cost and pattern) under cost function.

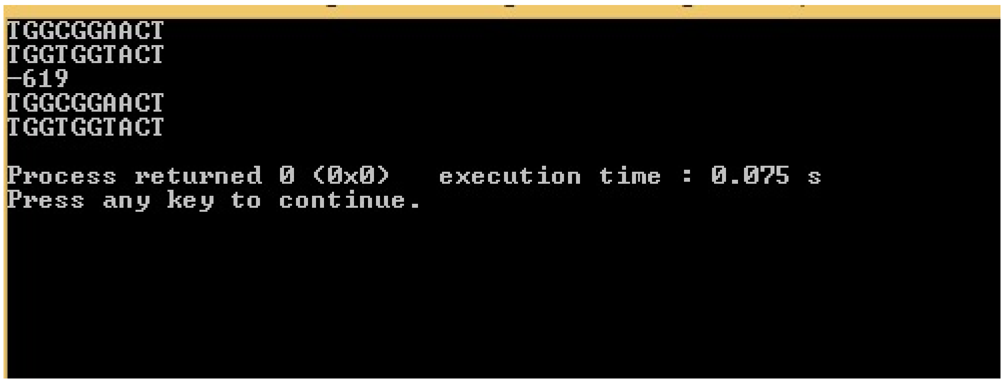
$$\gamma(k) = \delta + (k - 1) \times \beta$$

Figure 3.1: Gap penalty function

- β is gap opening penalty
- β is gap extension penalty
- k is gap spread

3.1 Output Format

- compile the code and run executable file(in case of windows)/a.out file(in case of unix)
- Enter target string(source string/first string) in terminal window and hit ENTER
- Now you are in new line, enter query string (second string) hit enter
- You will see output in the following format
- optimal cost value
- new target string
- new query string



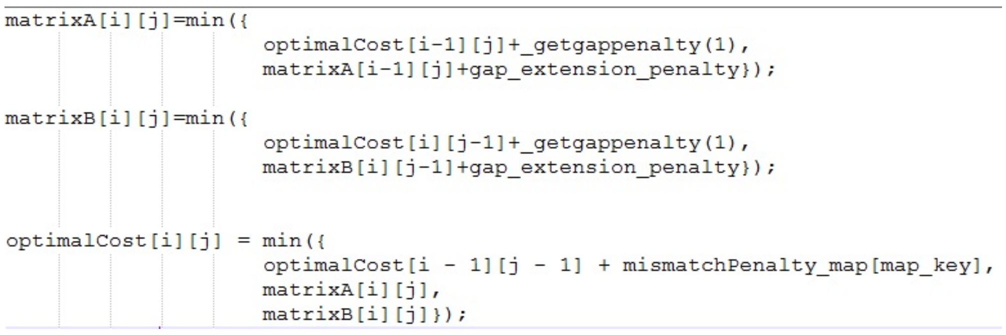
```
TGGCGGAACT
TGGTGGTACT
-619
TGGCGGAACT
TGGTGGTACT

Process returned 0 (0x0)   execution time : 0.075 s
Press any key to continue.
```

Figure 3.2: Output Format

3.2 Solution/Strategy

(a) we employ **Dynamic Programming paradigm** to solve this problem, namely gohot's algorithm for linear affine gap

(b) The corresponding recurrence relations is given by

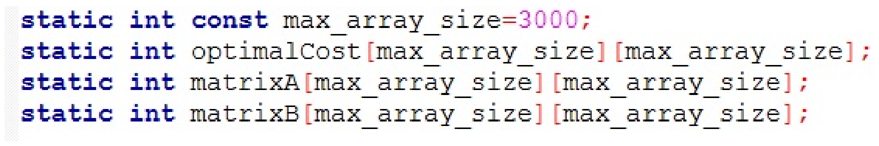


```
matrixA[i][j]=min({
                    optimalCost[i-1][j]+_getgappenalty(1),
                    matrixA[i-1][j]+gap_extension_penalty});

matrixB[i][j]=min({
                    optimalCost[i][j-1]+_getgappenalty(1),
                    matrixB[i][j-1]+gap_extension_penalty});


optimalCost[i][j] = min({
                    optimalCost[i - 1][j - 1] + mismatchPenalty_map[map_key],
                    matrixA[i][j],
                    matrixB[i][j]});
```

Figure 3.3: Recurrences for solution

(c) Three global matrices Optimal Cost,MatrixA and MatrixB being used, with maximum size 3000 x 3000 and a **global** variable **INFINITY** with a value of **25000** is used

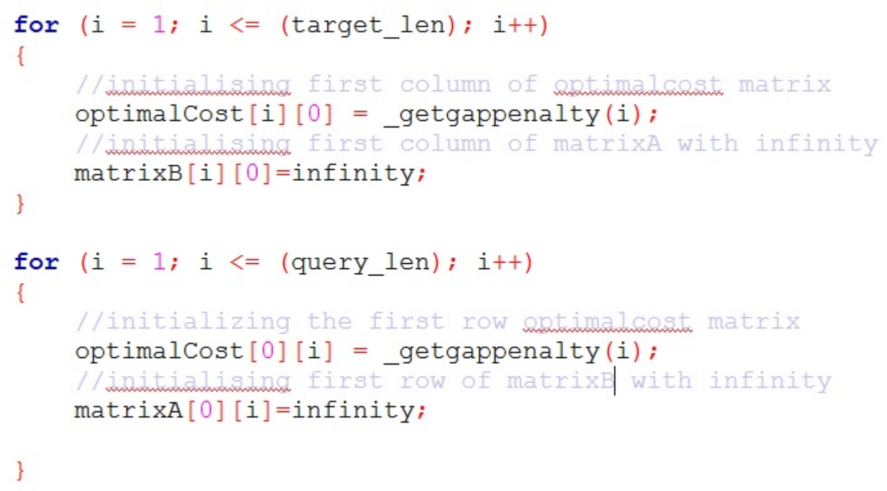
(d) Code can handle any DNA string with length at most **3000**



```
static int const max_array_size=3000;
static int optimalCost[max_array_size][max_array_size];
static int matrixA[max_array_size][max_array_size];
static int matrixB[max_array_size][max_array_size];
```

Figure 3.4: Optimal Cost Matrix

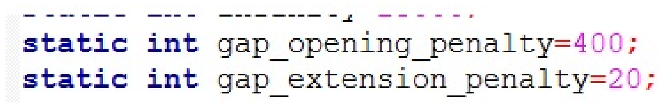
(e) Matrices were initialized as follow



```
for (i = 1; i <= (target_len); i++)
{
    //initialising first column of optimalcost matrix
    optimalCost[i][0] = _getgappenalty(i);
    //initialising first column of matrixA with infinity
    matrixB[i][0]=infinity;
}

for (i = 1; i <= (query_len); i++)
{
    //initializing the first row optimalcost matrix
    optimalCost[0][i] = _getgappenalty(i);
    //initialising first row of matrixB with infinity
    matrixA[0][i]=infinity;

}
```

Figure 3.5: Matrices initialization

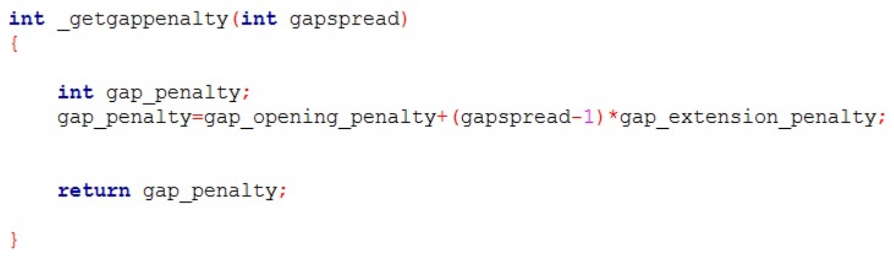
(f) The Corresponding penalties per the problem description used in the code



```
static int gap_opening_penalty=400;
static int gap_extension_penalty=20;
```

Figure 3.6: Initialization of δ and β

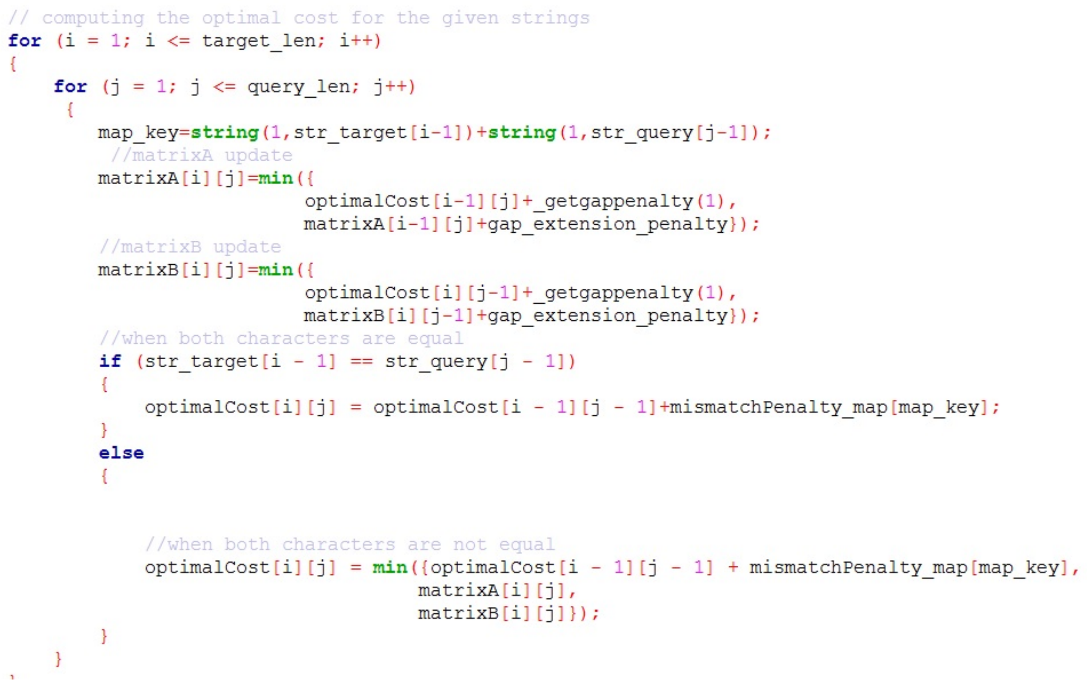
(g) The **get Gap Penalty()** function was employed to calculate the gap penalty



```
int _getgappenalty(int gapspread)
{

    int gap_penalty;
    gap_penalty=gap_opening_penalty+(gapspread-1)*gap_extension_penalty;


    return gap_penalty;

}
```

Figure 3.7: Gap Penalty Function

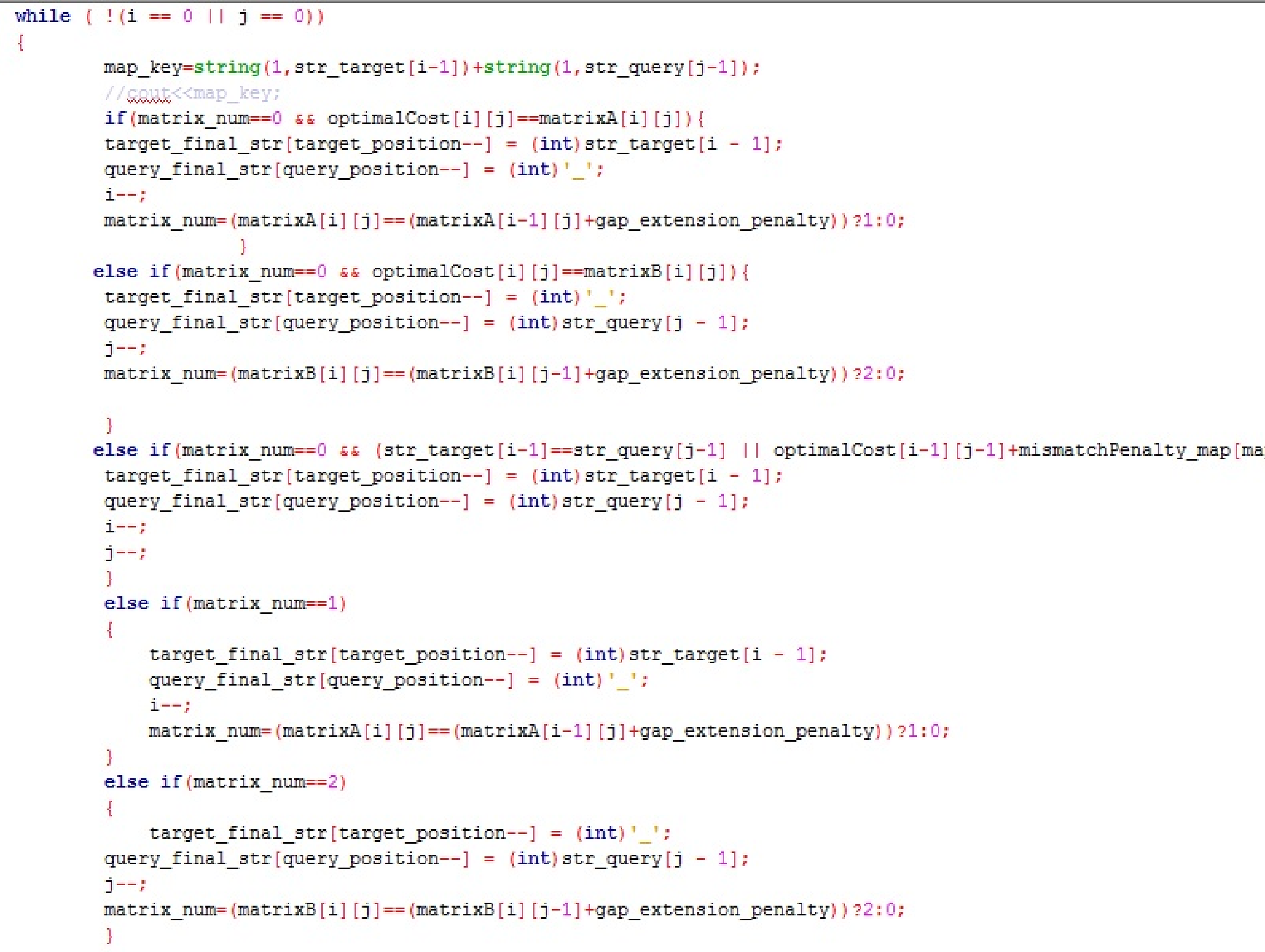
(h) Optimal cost,matrixA and matrixB matrices are filled using corresponding recurrences as follow



```
// computing the optimal cost for the given strings
for (i = 1; i <= target_len; i++)
{
    for (j = 1; j <= query_len; j++)
     {
        map_key=string(1,str_target[i-1])+string(1,str_query[j-1]);
         //matrixA update
        matrixA[i][j]=min({
                          optimalCost[i-1][j]+_getgappenalty(1),
                          matrixA[i-1][j]+gap_extension_penalty});
        //matrixB update
        matrixB[i][j]=min({
                          optimalCost[i][j-1]+_getgappenalty(1),
                          matrixB[i][j-1]+gap_extension_penalty});
        //when both characters are equal
        if (str_target[i - 1] == str_query[j - 1])
        {
            optimalCost[i][j] = optimalCost[i - 1][j - 1]+mismatchPenalty_map[map_key];
        }
        else
        {


            //when both characters are not equal
            optimalCost[i][j] = min({optimalCost[i - 1][j - 1] + mismatchPenalty_map[map_key],
                                    matrixA[i][j],
                                    matrixB[i][j]});
        }
    }
```

Figure 3.8: Updating matrices

(i) Once the matrices were filled, we use **back tracking** to print the output



```
while ( !(i == 0 || j == 0))
{
        map_key=string(1,str_target[i-1])+string(1,str_query[j-1]);
        //cout<<map_key;
        if(matrix_num==0 && optimalCost[i][j]==matrixA[i][j]){
        target_final_str[target_position--] = (int)str_target[i - 1];
        query_final_str[query_position--] = (int)'_';
        i--;
        matrix_num=(matrixA[i][j]==(matrixA[i-1][j]+gap_extension_penalty))?1:0;
                  }
       else if(matrix_num==0 && optimalCost[i][j]==matrixB[i][j]){
        target_final_str[target_position--] = (int)'_';
        query_final_str[query_position--] = (int)str_query[j - 1];
        j--;
        matrix_num=(matrixB[i][j]==(matrixB[i][j-1]+gap_extension_penalty))?2:0;

        }
       else if(matrix_num==0 && (str_target[i-1]==str_query[j-1] || optimalCost[i-1][j-1]+mismatchPenalty_map[ma
        target_final_str[target_position--] = (int)str_target[i - 1];
        query_final_str[query_position--] = (int)str_query[j - 1];
        i--;
        j--;
        }
        else if(matrix_num==1)
        {
            target_final_str[target_position--] = (int)str_target[i - 1];
            query_final_str[query_position--] = (int)'_';
            i--;
            matrix_num=(matrixA[i][j]==(matrixA[i-1][j]+gap_extension_penalty))?1:0;
        }
        else if(matrix_num==2)
        {
            target_final_str[target_position--] = (int)'_';
        query_final_str[query_position--] = (int)str_query[j - 1];
        j--;
        matrix_num=(matrixB[i][j]==(matrixB[i][j-1]+gap_extension_penalty))?2:0;
        }
```

Figure 3.9: Back Tracking

3.3 Correctness of the Algorithm

Dynamic Programming uses recurrences, constitutes optimal solutions. so correctness focuses on proving recurrence relations are **correct**. This part is already covered in class so skipping it.

3.4 Efficiency of the Algorithm

3.4.1 Space Complexity

(j) Size of the Optimal Cost array = $O(mn)$

(k) Size of the MatrixA array = $O(mn)$

(l) Size of the MatrixB array = $O(mn)$

(m) n = length of the Target DNA sequence

(n) m = length of the Query DNA sequence

(o) Additional Space for storing DNA sequences = $O(m+n)$

(p) Memory requirements for auxiliary variables = $O(c)$

(q) c=constant space

(r) Total Space Complexity $= O(3mn+m+n+c) = O(mn)$

3.4.2 Time Complexity

(s) Time for Initialization(first column,first row) of the **matrices** is $O(m+n)$

(t) Computation for each cell of the optimal cost matrix is 3 cases (constant)

(u) Computation for each cell of the MatrixA matrix is 2 cases (constant)

(v) Computation for each cell of the MatrixB matrix is 2 cases (constant)

(w) Total number of cells=$O(mn)$

(x) n = length of the Target DNA sequence

(y) m = length of the Query DNA sequence

(z) Time for back tracking =$O(m+n)$

() Total Space Complexity $= O(m+n+3mn+2mn+2mn+m+n) = O(mn)$