November 12, 2019

Report on Assignment -2 (DNA Sequence Alignment)

Submitted by

Suresh Kumar Amalapuram

Roll No. CS19RESCH11001

AdvancedDataStructures & Algorithms

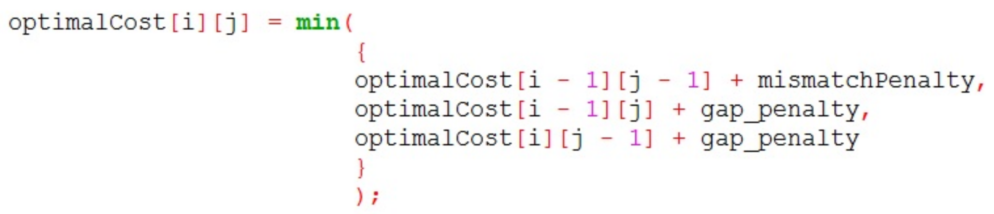
Problem 1

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

0.1 Solution/Strategy

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

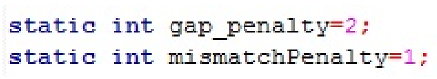
2. The corresponding recurrence relations is given by



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

Figure 1: Recurrence for solution

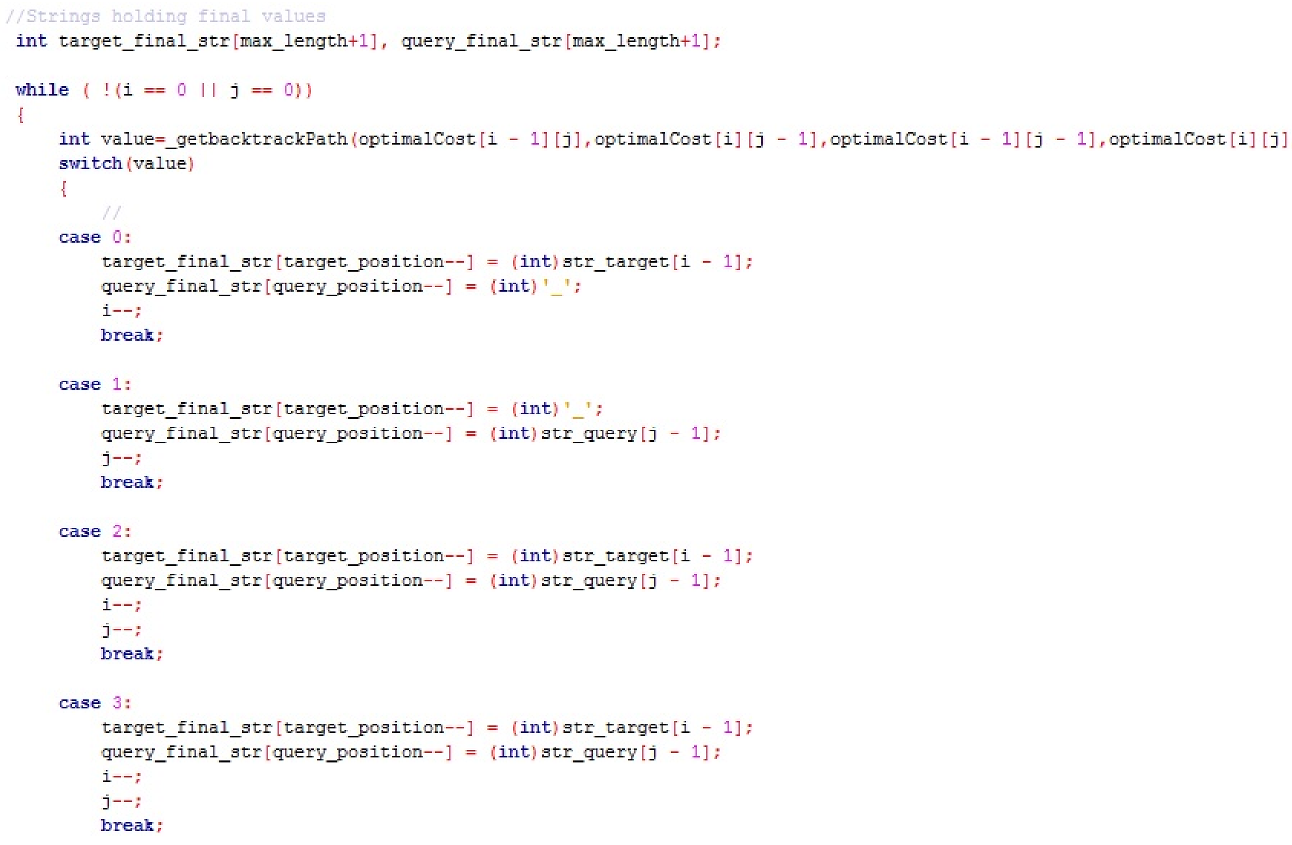
3. The Corresponding penalties per the problem description used in the code



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

Figure 2: Penalties

4. 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]
     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 3: Back Tracking

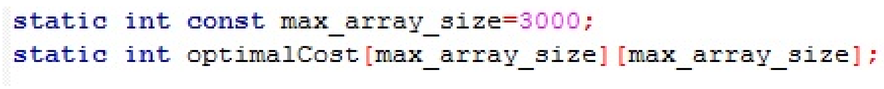
0.2 Correctness of the Algorithm

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.

0.3 Efficiency of the Algorithm

0.3.1 Key notes

- A global Optimal Cost array was created, with maximum size 3000 x 3000
- 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];
```

Figure 4: Optimal Cost Matrix

0.3.2 Space Complexity

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

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

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

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

- c=constant space

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

_

Description of each Function in Code

0.3.3 int _getnodePriority()

- Randomly assigns priorities to the Keys

0.3.4 tnode* _findRootNode(tnode* p_root)

- Returns Root of the Treap with p_root is one of its element

0.3.5 void _calculateNodeSize

- calculates current node size based on its left and right child

0.3.6 void _adjustAncestorNodeSize(tnode* p_node)

- Adjusts Node sizes of all nodes in the path from from current node to root of the treap

0.3.7 tnode* _getTreapMaxValue(tnode* p_node)

- Returns a pointer to the node with maximum value in treap rooted with p_node

0.3.8 tnode* _getTreapMinValue(tnode* p_node)

- Returns a pointer to the node with minimum value in treap rooted with p_node

0.3.9 tnode* _leftRotateforInsert(tnode* p_root)

- This function is invoked after inserting a element in the treap, to maintain heap property

0.3.10 tnode* _RightRotateforInsert(tnode* p_root)

- This function is invoked after inserting a element in the treap, to maintain Treap structure property (Binary search Tree + Heap property)

0.3.11 tnode* _leftRotateforDelete(tnode* p_root)

- This function is invoked after deleting a element in the treap, to maintain Treap structure property (Binary search Tree + Heap property)

0.3.12 tnode* _RightRotateforDelete(tnode* p_root)

- This function is invoked after deleting a element in the treap, to maintain Treap structure property (Binary search Tree + Heap property)

0.3.13 tnode* _validateHeapProperty(tnode* p_root)

- This function will ensure maintaining Treap structure property (Binary search Tree + Heap property) at p_root node

0.3.14 tnode* search_key(tnode *root,float key)

- This function will searches for the node whose value is key, returns NULL if not found

0.3.15 void inorder_printProxy(tnode *root)

- This is proxy method for inorder_print method

0.3.16 void inorder_print(tnode* root)

- This method will print in order sequence of the treap

0.3.17 tnode* _createNode(float p_key)

- creates new node with key value

0.3.18 tnode* insert_keyProxy(tnode *root, float k)

- This is an proxy method inserting an element into the treap, similar to binary search tree insertion

0.3.19 tnode* insert_key(tnode * & root, float k)

- This method will insert an element into treap and may apply multiple rotations (from bottom to top) to maintain Treap structure property (Binary search Tree + Heap property)

- This method will adjust all the node sizes of the ancestors nodes from newly inserted node to the root of the treap

- Validates the Heap property

0.3.20 tnode* _delete_nodeproxy(tnode *x)

- This is an proxy method for deleting an element from treap, similar to binary search tree deletion. may apply multiple rotations (from top to bottom) to maintain Treap structure property (Binary search Tree + Heap property)

- This method will adjust all the node sizes of the ancestors nodes from deleted node to the root of the treap

- Validates the Heap property

0.3.21 tnode* delete_node(tnode *root, tnode *x)

- This method invokes proxy delete method to perform deletion on treap.

0.3.22 tnode* join_treaps(tnode *root1, tnode *root2)

- k $\longleftarrow$ (maxkey(root1)+minkey(root2))/2
- create a new root with the key k
- Root1 will be the left sub-tree of k and Root2 will be the right sub-tree
- Delete the node k

0.3.23 tnode* successor(tnode *x))

- Similar to binary search tree successor finding

0.3.24 tnode* predecessor(tnode *x)

- Similar to binary search tree Predecessor finding

0.3.25 tnode* find_next(tnode* root, float k)

- Returns the smallest key larger than the key k, uses recursive procedure

0.3.26 tnode* find_previous(tnode *root, float key)

- Returns the largest key smaller than the key k, uses recursive procedure

0.3.27 int num_less_than(tnode *root,float k)

- Returns Number of nodes in the treap rooted at root, with keys less than k

0.3.28 tnode* _split_treapProxy(tnode *root,tnode *node,bool keynotfound)

- This is an proxy method for treap split, it is similar to typical binary search tree split

0.3.29 tnode* split_treap(tnode *root, float k))

- Searches for the key K in the treap with rooted root, if k found then invokes for proxy treap split
- if k not found in the treap, we insert key k with high priority, whihc will becomes the new root node and invoke the proxy split method.