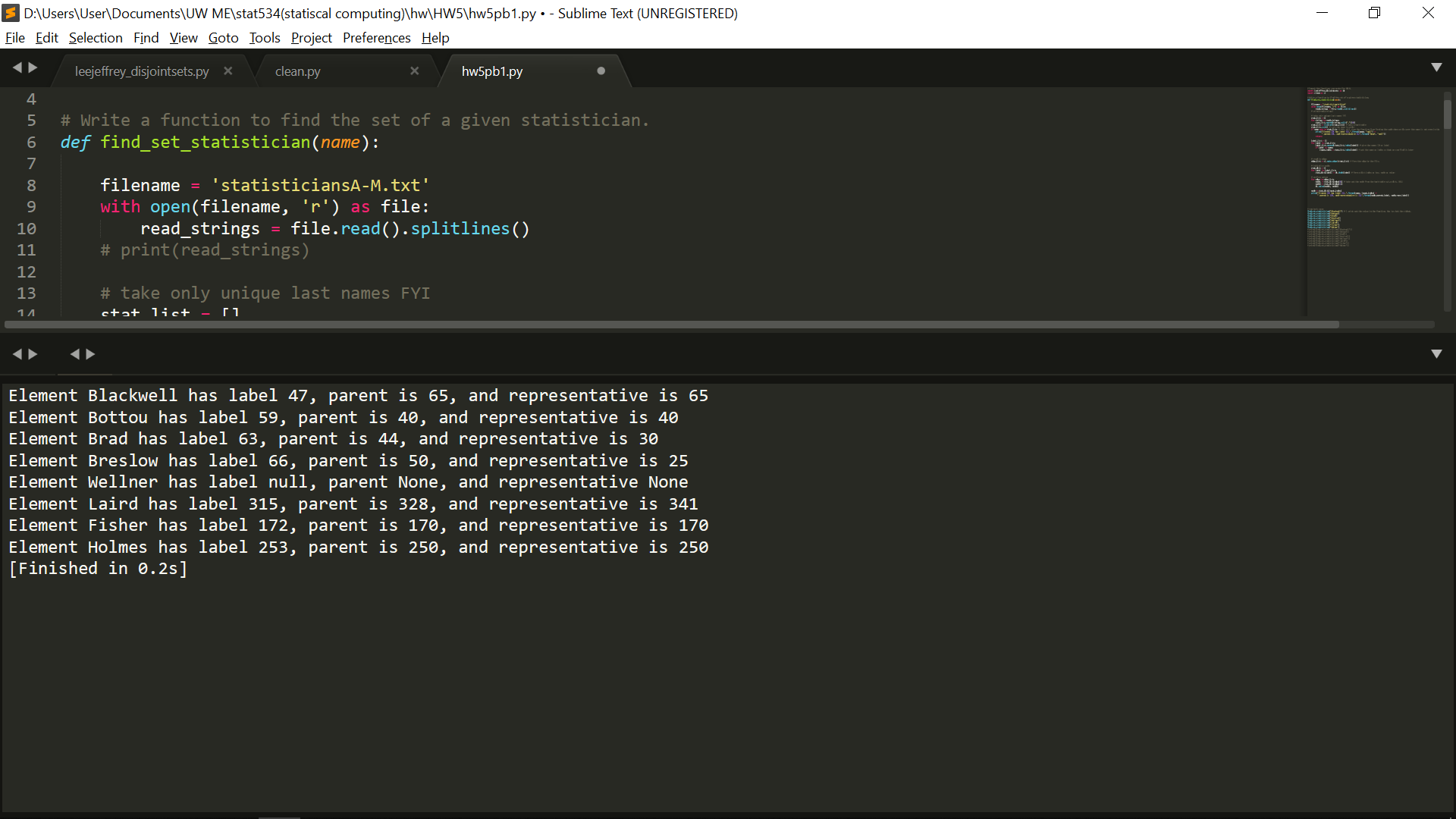
Statistical computing HW5

Jeffrey Lee

UWNetID: 1826649

Problem 1

(a)



(b)

Last time I implement as list to form the nodes. This will cause a problem since I have to check if the value is inside the list. This is acceptable for a small sample like last time since it’s still polynomial O(n) to check the list. This time I implemented as dictionary and set the key as index, value as nodes. Also, I set a variable to store the name index when reading through the file. This way, I can take out the node data easily by simply use the index in the dictionary, which will only take O(1).

Make-Set: This is for initializing the node as a set when it first comes out. I simply set its parent and representative as itself. Rank is 0. This will take O(1) to initialize.

Find\_Set: This function is for finding the representative of the set; therefore, when it’s not the representative, it will recursively call the function to find it.

Worst case is when it form a super unbalance tree and will take O(n) to climb to the top.

Union: This is to form the tree by link the representatives. Therefore, we have to find the representative by Find\_Set function and link it together. In link function, we compare the rank of the representatives, the node with larger rank becomes the parent. When the rank is equal, we arbitrary choose a node as parent. In my implementation, I choose node2 as node1’s parent, which means node1 points to node2. Those comparison of values and assignment operation will take O(1). Finally, we return the union and that’s how we get the forest.

Problem 2

(a)

(b) Without twiddle function

Since the cost of copy and replace is 0 and 1 respectively, using copy when match, and using replace when not match can sometimes give us the optimal solution. And since the cost of delete and insert are both 2, there’s no difference to choose delete or insert when the length of text and target are different. In my pseudo, I used insert when the final solution does not match the target.

**Pseudo:**

X = [X1, X2, X3….Xm] # text

Y = [Y1, Y2, Y3…..Yn] # target

Z = [Z1, Z2, Z3…..Zk] # array to store

string \*i = X; // set a pointer to point to the first value of X array

string \*j = Y; // set a pointer to point to the first value of Y array

string \*k = Z; // set a pointer to the stored array

for (int i = 0; i < X.length(); i++)

if X[i] == Y[j]

Z[k] = copy(X[i]); // copy the value from X[i]

i++;

j++;

k++;

elif X[i] != Y[j]

Z[k] = replace(Y[j]) // replace by the value of Y[j]

i++;

j++;

k++;

elif j > Y.length()

kill // kill the loop when out of range of Y.

// this will happen when text is longer than target.

return Z

if (Y.length() != Z.length()) // we have check the text longer case, now check target // longer case

Z[k] = insert(Y[j]) // insert the value of Y[j] to get to the target.

k++;

j++

return Z

**Runtime:** we have run through the X in worst case therefore for the loop is O(m). If it’s still not enough to reach the target, we will also need to fill the alphabet by inserting, this will cause O(n-m). Therefore, the total runtime will be O(n)

**Space:** We need to store the value for the X, Y and Z. In X and Y is cost static memory in the beginning and will store in the stack which will cost O(m+n). The Z memory will grow dynamically as the word expand and will store in the heap. Eventually, it will cost O(k). Therefore, the total memory cost will be O(m+n+k)

(c) With twiddle function

The cost of twiddle is 1, which is the same as cost of replace. However, when we find the situation that the target is in reverse order of text, we only need to perform twiddle once instead replace twice. If we want to know when to use twiddle function, we need to look forward then the current location, to check if the adjacent alphabet is in reverse-ordered. With that in mind, here’s my pseudo.

**Pseudo:**

X = [X1, X2, X3….Xm] # text

Y = [Y1, Y2, Y3…..Yn] # target

Z = [Z1, Z2, Z3…..Zk] # array to store

string \*i = X; // set a pointer to point to the first value of X array

string \*j = Y; // set a pointer to point to the first value of Y array

string \*k = Z; // set a pointer to the stored array

for (int i = 0; i < X.length(); i++)

if X[i] == Y[j]

Z[k] = copy(X[i]); // copy the value from X[i]

i++;

j++;

k++;

elif X[i] != Y[j]

if (X[i+1] == Y[j] && X[i] == Y[j+1])

twiddle(X[i], X[i+1])

Z[k] = copy(X[i])

Z[k+1] = copy(X[i+1])

K = k +2;

i = i + 2;

j = j + 2;

else

Z[k] = replace(Y[j]) // replace by the value of Y[j]

i++;

j++;

k++;

elif j > Y.length()

kill // kill the loop when out of range of Y.

// this will happen when text is longer than target.

return Z

if (Y.length() != Z.length()) // we have check the text longer case, now check target // longer case

Z[k] = insert(Y[j]) // insert the value of Y[j] to get to the target.

k++;

j++

return Z

**Runtime:** This is the same as the previous one. However, it will be a slightly different when we perform twiddle instead of replace. That cost difference is not much and can be ignored. In the worst case, we still need to run through the whole text, taking O(m) and inserting for the rest, taking O(n-m). Total will take O(n)

**Space:** It will be the same as part b, taking O(n+m+k)