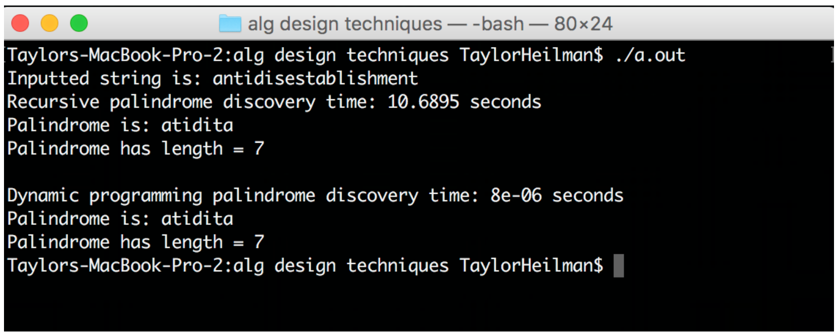
Algorithm Design Techniques

Proficiency: Mastery

In **Project 1001** we were assigned the task of implementing an algorithm which found the longest palindrome subsequence in a string. For this project I showed the ability to not only design algorithms using divide & conquer techniques but also design algorithms using dynamic programming. To show the advantages of dynamic programming I implemented a recursive function to find the longest palindrome subsequence in a string. While this technique showed my ability to design recursive functions the actual function was severely inefficient. The recursive function computed the same problems multiple times, causing a poor run time. To improve efficiency I implemented a dynamic programming algorithm to compute longest palindrome subsequence. Unlike its recursive counterpart, the dynamic programming function used the problem's structure to its advantage. Instead of computing the same subproblems over and over again, the dynamic function used answers to smaller subproblems to formulate the answers to larger sub problems. This bottom up technique allowed for a far better runtime than the recursive solution.

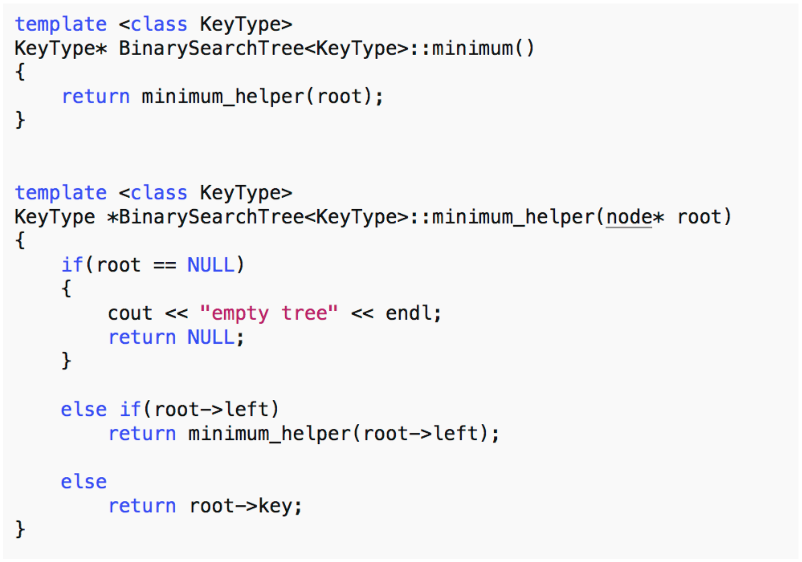


```
Taylors-MacBook-Pro-2:alg design techniques TaylorHeilman$ ./a.out
Inputted string is: antidisestablishment
Recursive palindrome discovery time: 10.6895 seconds
Palindrome is: atidita
Palindrome has length = 7

Dynamic programming palindrome discovery time: 8e-06 seconds
Palindrome is: atidita
Palindrome has length = 7
Taylors-MacBook-Pro-2:alg design techniques TaylorHeilman$
```

When compared on a Mac Book Pro running OS X 10.11.4 the efficiency of the dynamic programming function was far superior than the recursive function. To compute the longest palindrome subsequence for a twenty letter string it took the recursive function 10.6 seconds, while it took the dynamic programming function just 0.0000008 seconds. Attached is the write up from **Project 1001** showing how I developed the dynamic function using a recurrence that defines the value of an optimal solution as well as the code. From this recurrence I wrote pseudocode for the algorithm and finally I implemented the pseudocode for the final dynamic programming function.

In **Project 0110** I had to implement a binary search tree. Due to the structure of a binary search tree, the operations used in the class could be mostly implemented using recursion. Because of this I had to master the divide and conquer technique. When looking at the structure of a binary search tree it became obvious that you can find a desired answer by dividing the problem into smaller problems. An example of this is shown in the simple Minimum() function below. The Minimum() function is used to



```
template <class KeyType>
KeyType* BinarySearchTree<KeyType>::minimum()
{
    return minimum_helper(root);
}


template <class KeyType>
KeyType *BinarySearchTree<KeyType>::minimum_helper(node* root)
{
    if(root == NULL)
    {
        cout << "empty tree" << endl;
        return NULL;
    }

    else if(root->left)
        return minimum_helper(root->left);

    else
        return root->key;
}
```

find the minimum key in the binary search tree. By definition of a binary search tree, all of keys in the left subtree of ‘x’ must be less than or equal to the key of ‘x’, hence the minimum key in the binary search tree is found by traversing left from the root until a node with no left child is found.   This is easily done with recursion. We start at the root of the tree and check if the current node has a left child. This is our divide step. We divide the overall problem of finding the left most element in the binary search tree into the smaller subproblem of whether a node has a left child or not. Using the outcomes of these smaller subproblems we eventually conquer the overall problem. If the current node has a left child we recursively call Minimum() on the left child. This step is repeated until we reach a node with no left child and return the current node. Attached are other recursive functions used to implement other functions in the binary search tree class.

Name: Taylor Heilman and Kevin Benson

Mon. May 2

CS 271: Project 1001

1. a.) The structure of an optimal solution to $f(m,n)$ would be a palindromic string of max length.

   b.) $f(m,n)$ represents the length of the longest palindrome subsequence from m^{th} character to the n^{th} character

$$f(m,n)\begin{cases} 1, & m=n \\ f(m+1,n-1)+2, & a_m=a_n \\ max[f(m+1,n),f(m,n-1)] & else \end{cases}$$

   c.)

   Create a square array with dimensions ixj where both i and j are equal to the length of the input string.

   Make the diagonal indices = 1 (since a string of length 1 is a palindrome of itself) and all other indices = 0.

   Iterate through all columns and examine every item $c[i][j]$ above the diagonal. If the i^{th} and j^{th} characters in the input string are identical, the solution to subproblem f(i,j) is two greater than the solution to subproblem f(i+1, j-1), which is stored at index [i-1][j+1].

   If this is not the case, the value of the solution is the greater of either the item to the left or the item below the item in question.

   Our c++ implementation of this algorithm is provided below.

```
//================================================================================
// Kevin Benson and Taylor Heilman
// May 2, 2016
// dynamic.cpp
//
// This file contains both the recursive and dynamically-programmed solutions
// to problem 15.2 of the textbook.
//================================================================================
#include <string.h>
#include <iostream>
#include<stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <sys/time.h>
using namespace std;
//============================================
// reverse helper function
//============================================
string reverse(string s)
{
        int n = s.length();
        if (n == 0)
                return string("");
        return s[n - 1] + reverse(s.substr(0, n - 1));
}
//============================================
// max helper function
//============================================
int max (int x, int y) {
        if(x > y)
                return x;
        else
                return y;
}
//============================================
// Recursive palindrome function
// Returns the length of the longest palindromic subsequence in s
//============================================
string palindrome(string s, int i, int j){
        if(i==j){
                string c;
                c += s[i];
                return c;
        }
        if(i+1 == j and s[i] == s[j]){
                string  toreturn;
                toreturn = toreturn + s[i];
                toreturn = toreturn + s[j];
                return toreturn;
        }
        if(s[i] == s[j]){
                string toreturn;
                toreturn = toreturn + s[i];
                toreturn = toreturn + palindrome(s, i+1, j-1);
                toreturn = toreturn + s[j];
                return toreturn;
        }

        // Cannot use max to compare strings based on length, so do it the long way
        if((palindrome(s, i+1, j).length()) > (palindrome(s, i, j-1).length()))
                return palindrome(s, i+1, j);
        else
```

```
                return palindrome(s, i, j-1);
}

//============================================
// Dynamic Programming palindrome function
// Returns the length of the longest palindromic subsequence in s
//============================================
string dynamicPalindrome(string str)
{
        int l = str.length();
        int c[l][l];      // value table, which is square as we're funcitonally comparing two
identical strings
        // *** INITIALIZE THE TABLES ***

        for (int i = 0; i < l; i++){
                for (int j = 0; j < l; j++){
                        c[i][j] = 0;
                }
        }
        // *** FILL DIAGONAL OF TABLE WITH ONES ***
        for (int i = 0; i < l; i++){
                c[i][i] = 1;
        }
        for (int i = 1; i < l; i++)
        {
                for (int j = i-1; j >=0; j--)
                {
                        if (str[i] == str[j]){
                                c[i][j] = c[i-1][j+1] + 2;
                        }
                        else
                        {
                                c[i][j] = max(c[i-1][j], c[i][j+1]);
                        }
                }
        }

        int i = l-1;
        int j = 0;

        // Solution to subproblem f(m,n) is at index [m][m-n], so our final
        // solution is at the top right of our table
        int palLength = c[l-1][0];

        // Backtrace our array to find the palindrome itself
        string palindrome;
        while(c[i][j] != 0){
                if(c[i][j] != c[i-1][j] and c[i][j] != c[i][j+1]){
                        palindrome = palindrome + str[i];
                        i--;
                        j++;
                }
                else if (c[i][j] == c[i-1][j])
                        i--;
                else
                        j++;
        }
        // Reconstruct palindrome based on the parity of its length
        if(palLength % 2 == 0){
                palindrome = palindrome + reverse(palindrome);
        }
        else{
                for(int i = palindrome.length()-2; i >=0; i--){
                        palindrome += palindrome[i];
                }
        }

        return palindrome;
}

int main(){
        string q = "antidisestablishment";
        int i = q.length();

        // Recursive solution time analysis
        long diffSeconds, diffUSeconds;
        timeval timeBefore, timeAfter;
        gettimeofday(&timeBefore, NULL);

        string pal = palindrome(q, 0, i-1);

        gettimeofday(&timeAfter, NULL);
        diffSeconds = timeAfter.tv_sec - timeBefore.tv_sec;
        diffUSeconds = timeAfter.tv_usec - timeBefore.tv_usec;

        cout << "Recursive palindrome discovery time: " << diffSeconds + diffUSeconds/1000000.
0 << " seconds" << endl;
        cout << "palindrome is: " << pal << endl;
        cout << "palindrome has length = " << pal.length() << endl << endl;

        // Dynamic solution time analysis
        gettimeofday(&timeBefore, NULL);

        pal = dynamicPalindrome(q);

        gettimeofday(&timeAfter, NULL);

        diffSeconds = timeAfter.tv_sec - timeBefore.tv_sec;
        diffUSeconds = timeAfter.tv_usec - timeBefore.tv_usec;

        cout << "dynamic programming palindrome discovery time: " << diffSeconds + diffUSecond
s/1000000.0 << " seconds" << endl;
        cout << "palindrome is: " << pal << endl;
        cout << "palindrome has length = " << pal.length() << endl;

        return 0;
}
```

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include "bst.h"
using namespace std;




//=========================================================================
// Insert()
// Use Recursion to Properly Insert an Item into the BST
//=========================================================================

template <class KeyType>
void BinarySearchTree<KeyType>::insert(KeyType *x)
{
        if (root)
        {
                return insert_helper(root, x);
        }


        root = new node<KeyType>(x);
}


template <class KeyType>
void BinarySearchTree<KeyType>::insert_helper(*root, KeyType* key)
{
        //create new node
        if (root->key >= key)
        {

                if (root->left)
                        return insert_helper(root->left,key);

                else
                        root->left = new node<KeyType>(key);

        }

        else
        {
                if(root->right)
                {
                        return insert_helper(root->right, key);
                }

                else
                {
                        root->right = new node<KeyType>(key);
                }
        }
}

//========================================================================
// Get()
// Use Recursion to Properly Return First Element with Key Equal to k
//========================================================================

template <class KeyType>
KeyType *BinarySearchTree<KeyType>::get(const KeyType& x)
{
        return search_helper(root,key);
}
```

```
template <class KeyType>
KeyType *BinarySearchTree<KeyType>::get_helper(node* root, const KeyType& key)
{
        if(root->key == key)
                return root;

        else
        {
                if(root->key > key)
                {
                        root = root->right;

                        if(root == NULL)
                        {
                                cout << "not in tree" << endl;
                                return -1;
                        }

                        else
                                return get_helper(root,key);
                }

                else
                {
                        root = root->left;
                        if(root == NULL)
                        {
                                cout << "not in tree" << endl;
                                return -1;
                        }

                        else
                                return get_helper(root,key);

                }
        }
}



//========================================================================
// Minimum()
// Use Recursion to Properly Return the Minimum Element in the BST
//========================================================================

template <class KeyType>
KeyType* BinarySearchTree<KeyType>::minimum()
{
        return minimum_helper(root);
}


template <class KeyType>
KeyType *BinarySearchTree<KeyType>::minimum_helper(node* root)
{
        if(root == NULL)
        {
                cout << "empty tree" << endl;
                return NULL;
        }

        else if(root->left)
                return minimum_helper(root->left);

        else
                return root->key;
}

//========================================================================
// Maximum()
// Use Recursion to Properly Return the Maximum Element in the BST
//=====================================================================

template <class KeyType>
KeyType* BinarySearchTree<KeyType>::maximum()
{
        return maximum_helper(root);
}


template <class KeyType>
KeyType *BinarySearchTree<KeyType>::maximum_helper(node* root)
{
        if(root == NULL)
        {
                cout << "empty tree" << endl;
                return NULL;
        }

        else if(root->right)
                return maximum_helper(root.right);
        else
                return root->key;
}


//=====================================================================
// inOrder()
// Use Recursion to Properly Return String of Elements From an Inorder Traversal
//=====================================================================


template <class KeyType>
void BinarySearchTree<KeyType>::inOrder()
{
        return inOrder_helper(root);
}


template <class KeyType>
void BinarySearchTree<KeyType>::inOrder_helper(node* root)
{
        if( root == NULL)
        {
                cout << "empty tree" << endl;
                return -1;
        }

        inOrder_helper(root->left);
        cout << root << key;
        inOrder_helper(root->right);
}




//=====================================================================
// preOrder()
// Use Recursion to Properly Return String of Elements From an Preorder Traversal
//=====================================================================



template <class KeyType>
void BinarySearchTree<KeyType>::preOrder()
{
        return preOrder_helper(root);
}
```

```
template <class KeyType>
void BinarySearchTree<KeyType>::preOrder_helper(node* root)
{
        if( root == NULL)
        {
                cout << "empty tree" << endl;
                return -1;
        }

        cout << root << key;
        preOrder_helper(root->left);
        preOrder_helper(root->right);
}


//========================================================================
// inOrder()
// Use Recursion to Properly Return String of Elements From an Postorder Traversal
//========================================================================


template <class KeyType>
void BinarySearchTree<KeyType>::postOrder()
{
        return postOrder_helper(root);
}


template <class KeyType>
void BinarySearchTree<KeyType>::postOrder_helper(node* root)
{
        if( root == NULL)
        {
                cout << "empty tree" << endl;
                return -1;
        }

        postOrder_helper(root->left);
        postOrder_helper(root->right);
        cout << root << key;
}


//========================================================================
// Predecessor()
// Use Recursion to Properly Return the Predecessor of x
//========================================================================

template <class KeyType>
KeyType* BinarySearchTree<KeyType>::predecessor(const KeyType& x)
{
        predecessor_helper(x, root);
}


template <class KeyType>
KeyType* BinarySearchTree<KeyType>::predecessor_helper(const KeyType& x, node* root)
{
        if (x->left)
                return maximum_helper(x->left);

        else
        {
                struct node *p = x->parent;
                while( p != NULL && x == p->left)
                {
                        x = p;
                        p = p->parent;
                }
                return p;

        }
}

//======================================================================
// Successor()
// Use Recursion to Properly Return the Successor of x
//======================================================================


template <class KeyType>
KeyType* BinarySearchTree<KeyType>::successor(const KeyType& x)
{
        successor_helper(x, root);
}


template <class KeyType>
KeyType* BinarySearchTree<KeyType>::successor_helper(const KeyType& x, node* root)
{
        if (x->right)           //x has a right child
                return minimum_helper(x->right);

        else                                    // successor is above x in tree
        {
                struct node *p = x->parent;
                while( p != NULL && x == p->right)
                {
                        x = p;
                        p = p->parent;
                }

                return p;
        }
}
```