Eric Paulz
CPSC 3120

Homework #1

Divide and Conquer

The approach that I took follows from the Matrix Chain Multiplication examples we saw in class, but also takes advantage of string functionality. MinPal() is a recursive function that works from the top down, breaking the problem down into smaller problems until they become trivial, and then building the answers back up to find the total result. First, we check to see if the string is either empty or a palindrome itself (any string of length 1 is a palindrome). If either is true, 0 is returned as there are no more partitions to be made. If this is not the case, we start by setting the maximum number of 'cuts' equal to the length of the string minus 1. Note that 'cuts' represents the number of partitions required to separate the string into only palindromes, so a string of all unique characters will always have n-1 partitions. Next, loop through the string from 1 through the string's length, each time calling MinPal() recursively for all substrings and recalculating the minimum number of 'cuts' that represent a string of only palindromes. Return the minimum number of partitions and then add 1 to the returned value, since we are looking for the minimum number of actual palindromes that make up the string. Following is the recursive formulation I used:

$$\text{MinPal(S)} = \begin{cases} 0 & \text{if n} = 0 \text{ or S is a palindrome} \\ \sum_{i=1}^{n} (MinPal(S.substring(0,i)) + MinPal(S.substring(i,n)) + 1) & \text{if S is not a palindrome and n} > 0 \end{cases}$$

This algorithm runs in exponential time in terms of the length of the input string ($O(2^n)$). We come to this conclusion by first observing that MinPal() alone (before any recursive calls are considered) runs in linear time due the single 'for' loop that goes from 1 to the length of the input string. The next observation that we must make is that each recursive call of MinPal() breaks its problem down into two subproblems. So, each level of the recursion tree for this function will have $2k$ nodes, where k equals the depth of the current level of the tree. Therefore, we see that the divide and conquer method of solving this problem does in fact take exponential time $O(2^n)$. This method is very inefficient due the many overlapping problems that it comes across. There are many substrings that may be considered more than once, and each recursive call costs precious resources. This method is only good for very short strings.

Eric Paulz
CPSC 3120

PSEUDOCODE

```
// function to compute a minimum value
min(x,y)
 1  return (x < y ? x : y)

// function to determine if a string is a palindrome
isPalindrome(S)
 1  i = 0, j = S.length()-1
 2
 3  /* start by checking that the outermost letters match, then move
 4     both indexes inward until you've reached the middle */
 5  while(i < (S.length() / 2)
 6    if a[i] != a[j]
 7      return false
 8    i++
 9    j--
10
11  return true

MinPal(S)
 1  // base case of recursion:
 2  // check if string is empty or is a palindrome
 4  if S.length()==0 or isPalindrome(S)
 5    return 0
 6
 7  n = S.length() // length of the string
 8  cuts = n-1     // maximum number of partitions for any string
 9
10  /* loop through string, following the recursive formulation to find the
11     minimum number of cuts (substring(a,b) implies a substring of S
12     (S[a..b]))
13  */
14  for i = 1 to n
15    cuts = min(MinPal(S.substring(0,i))+MinPal(S.substring(i,n))+1, cuts)
16
17  return cuts
```

**Note: It is important to add a 1 to the returned value inside the main function that calls MinPal() to get the correct number of palindromes. My MinPal(S) actually computes the number of partitions needed to split the string up into only palindromes, so the number of actual palindromes will always be cuts + 1.

Eric Paulz
CPSC 3120

Dynamic Programming

The dynamic programming approach to solving this problem is, in my opinion, much less intuitive than the recursive method, but it is vastly more efficient once you start dealing with longer strings. Basically, we maintain two matrices C and P. Each element of C represents the number of partitions ('cuts') a substring of input string S requires to be comprised of only palindromes. So, C[0][0] is just the first letter of S, and C[0][n-1] is the entire string S. Elements in P are either true or false and they keep track of whether or not their corresponding elements in C are palindromes. We know that all strings of length 1 are palindromes, so C[0][0], C[1][1], …, C[n-1][n-1] all equal 0 since length 1 substrings require no partitions. Following this, P[0][0], P[1][1], …, P[n-1][n-1] all equal true. By comparing each element to its neighboring elements and keeping track of which are palindromes, and then returning the value held in C[0][n-1] (the minimum partitions for string S to be made up of only palindromes), we have a speedy and efficient solution to the problem.

We can see that the running time of this program is $O(n^3)$ by observing that we have one main 'for' loop containing two nested 'for' loops that must be entered so long as the length of string *s* (*n*) is greater than 1 (unless, of course, string *s* itself is a palindrome… but we will not assume the best case scenario). The first 'for' loop runs from 2 (*subStrLen*) through *n*, which can safely be approximated as *n* times. The second 'for' loop runs from 0 through *n-subStrLen*+1. Since *subStrLen* starts out at 2 and goes to *n*, it may be more practical to say that this loop runs from *n*-1 down to 1, which can also be safely approximated as *n* times. The third and final 'for' loop runs from *k* (which is set to equal *i*) through *j* which is set to equal *i*+*subStrLen*-1 (essentially, 1 through *n*). The three 'for' loops being nested and each running in linear time, along with the fact that all other operations in the program ('if/else' statements, assignments, etc.) run in linear time or better, shows us that this program will have a total running time of $O(n^3)$. This approach is much more efficient as it does not consider any overlapping problems. You can observe in the sample output below that the two approaches appear equal in the first few examples, but the dynamic program vastly outperforms the recursive one in the last few where the length of the string is longer.

** Program and sample output are found on the next few pages
(I have also attached a copy of my code file in case it is too small to read in the document)

Eric Paulz
CPSC 3120

```cpp
#include <iostream>
#include <string>
#include <climits>

int MinPal(std::string s);
int min(int x, int y);

int main(int argc, char* argv[]){

    // for usage
    if(argc != 2){
        std::cout << "Usage: <executable> <string>\n";
        return 1;
    }

    std::string s = argv[1];

    int x = MinPal(s);

    std::cout << "\nThe minimum number of palindromes in '" << s << "' is " << x+1;
    std::cout << std::endl << std::endl;

    return 0;
}

int min(int x, int y){
    return(x < y ? x : y);
}

int MinPal(std::string s){
    // get string length
    int strLen = s.length();
    int i, j, k, subStrLen;

    int C[strLen][strLen];     // min number of palindrome cuts in each substring
    bool P[strLen][strLen];  // true if substring is a palindrome

    // every substring of length 1 is a palindrome
    for(i = 0; i < strLen; i++){
        C[i][i] = 0;
        P[i][i] = true;
    }

    // consider all substrings of length subStrLen starting from 2 to strLen
    for(subStrLen = 2; subStrLen <= strLen; subStrLen++){
        // for substring of length subStrLen, check all different starting indexes
        for(i = 0; i < strLen-subStrLen+1; i++){
            j = i+subStrLen-1; // set ending index

            // if subStrLen is 2, only compare those two characters
            // otherwise, check those two characters AND the value of P[i+1][j-1]
            if(subStrLen==2)
                P[i][j] = (s[i] == s[j]);
            else
                P[i][j] = (s[i] == s[j]) && P[i+1][j-1];

            // if string[i..j] is a palindrome, then C[i][j] is 0 (minimum cuts is 0)
            if(P[i][j])
                C[i][j] = 0;
            else{
                // make a cut at every location starting from i to j-1
                // and compute the minimum cost cut
                C[i][j] = INT_MAX;
                for(k = i; k <= j-1; k++){
                    C[i][j] = min(C[i][j], C[i][k] + C[k+1][j] + 1);
                }
            }
        }
    }
    //C[0][strLen-1] represents minimum cuts over the whole original string... return it
    return C[0][strLen-1];
}
```

Eric Paulz
CPSC 3120

SAMPLE OUTPUT

```
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# g++ -Wall -o recursive minPal_recursive.cpp
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# g++ -Wall -o dynamic minPal_dynamic.cpp
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./recursive a

The minimum number of palindromes in 'a' is 1


real    0m0.026s
user    0m0.000s
sys     0m0.016s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./dynamic a

The minimum number of palindromes in 'a' is 1


real    0m0.049s
user    0m0.000s
sys     0m0.016s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./recursive aba

The minimum number of palindromes in 'aba' is 1


real    0m0.063s
user    0m0.000s
sys     0m0.031s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./dynamic aba

The minimum number of palindromes in 'aba' is 1


real    0m0.057s
user    0m0.000s
sys     0m0.031s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./recursive abc

The minimum number of palindromes in 'abc' is 3


real    0m0.050s
user    0m0.000s
sys     0m0.016s
```

```
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./dynamic abc

The minimum number of palindromes in 'abc' is 3


real    0m0.049s
user    0m0.000s
sys     0m0.031s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./recursive bobseesanna

The minimum number of palindromes in 'bobseesanna' is 3


real    0m0.065s
user    0m0.016s
sys     0m0.000s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./dynamic bobseesanna

The minimum number of palindromes in 'bobseesanna' is 3


real    0m0.053s
user    0m0.000s
sys     0m0.000s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./recursive bobseesannaandhannah

The minimum number of palindromes in 'bobseesannaandhannah' is 7


real    1m11.594s
user    1m10.188s
sys     0m0.063s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./dynamic bobseesannaandhannah

The minimum number of palindromes in 'bobseesannaandhannah' is 7


real    0m0.046s
user    0m0.000s
sys     0m0.000s
```

```
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./recursive abcdefghijklmnop

The minimum number of palindromes in 'abcdefghijklmnop' is 16


real    0m1.148s
user    0m1.078s
sys     0m0.031s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./recursive abcdefghijklmnopqrs

The minimum number of palindromes in 'abcdefghijklmnopqrs' is 19


real    0m31.138s
user    0m30.719s
sys     0m0.016s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./dynamic abcdefghijklmnop

The minimum number of palindromes in 'abcdefghijklmnop' is 16


real    0m0.055s
user    0m0.000s
sys     0m0.016s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./dynamic abcdefghijklmnopqrs

The minimum number of palindromes in 'abcdefghijklmnopqrs' is 19


real    0m0.046s
user    0m0.000s
sys     0m0.031s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./recursive racecarracecarracecarracecar

The minimum number of palindromes in 'racecarracecarracecarracecar' is 1


real    0m0.068s
user    0m0.000s
sys     0m0.016s
root@LAPTOP-NLCC5G3D:/mnt/c/Users/Eric/spring18/3120/hw1# time ./dynamic racecarracecarracecarracecar

The minimum number of palindromes in 'racecarracecarracecarracecar' is 1


real    0m0.048s
user    0m0.016s
sys     0m0.016s
```