**CS 180** Homework 5

**Problem 1**

1. This greedy algorithm for giving change would fail on many cases. For example if no-greedy country had coin denominations of $5, $8 and $11, using the greedy algorithm to spit out change for $21 would not work. It would give out $11 first, then $8 and we would be left with $2 in change and no coin denomination for that. Meanwhile, the ideal way would be to give out two $5 coins and one $11 coin.
2. This algorithm works because it employs dynamic programming and checks all possibilities of coins to use rather than greedily prioritizing the highest value ones. It solves previous subproblems from values of 1 up to the target value to get the final answer. At each iteration of the value, it checks it against every possible coin denomination to find the minimum number of coins needed.

// given array of coin denominations *v* in sorted order

generateChange(value m, denominations v)

int table[m+1]

table[0] = 0 // base case that m == 0

for i = 1; i <= m; i++

table[i] = INT\_MAX

for i = 1; i <= m; i++ // loop through 1 to target value m

for j = 0; j < m; j++ // loop through coins smaller than value i

if v[j] <= i

prev = table[i - v[j]]

if prev != INT\_MAX && prev + 1 < table[i] // if we have a solution for subproblem and it’s

// smaller than current solutions

table[i] = prev + 1;

return table[m];

**Problem 2**

Given the length *l* and probability *p* of being accessed for each file, we can just find solve *p* / *l* for each file. This would give us the probability of access per unit of length. Now that we’ve generalized the file length to one unit, we can just sort all the files by *p* / *l* and the result would be the optimal solution. No dynamic programming would be required for this and the solution would run in O(n log n) if we use an efficient sorting algorithm.

**Problem 3**

printWords(n, M, l)

for i = n; i >= 1; i--

p = i

rem = M - l[p]

while rem - l[p] > 0 && p < n

rem = rem - l[p]

p += 1

if p = n

then c[i] = 0

lineend[i] = n

else

c[i] = INT\_MAX

sum\_lk = 0

for j = i to p

sum\_lk = sum\_lk + lj

cost = ( M -j + i - sum\_lk )3 + c[j+1]

if cost < c[i]

then c[i] = cost

lineend[i] = j

start = 1

while start <= n

for word = start to lineend[start]

print ln

print newline

start = lineend[start] + 1

**Problem 4**

longestPalindromicSubsequence(string s)

size = s.length()

int dp[][] = new int[size][size] // cell values default to 0

for i = 0; i < size; i++

dp[i][i] = 1 // initialize diagonal cells to 1, every character is a palindrome of itself

for len = 2; len <= size; len++

for start = 0; start <= size - len; start++

end = start + len - 1

if s[start] == s[end] && len == 2 // if equal and we’re looking for palindromes of 2

dp[start][end] = 2

else if s[start] == s[end] // if equal and longer than length 2

dp[start][end] = dp[start+1][end-1] + 2

else // if not equal

dp[start][end] = max(dp[start+1][end], dp[start][end-1])

return dp[0][size-1]

**Problem 4**

In this algorithm, I used a dynamic programming approach by initializing a boolean table of n x n values where n is the length of the input string. If a substring from index i to j of the string is a palindrome, the cell at table[i][j] would be set to true. So we start off by setting the diagonals [0][0], [1][1], etc. to true (since a one character string is a palindrome of itself) and the other cells to false. Then we check for two characters in a row to count for even palindrome cases. Next, we check for palindromes of length 3 and higher by consulting our table to make sure the inner substring is already a palindrome before we add the new substrings.

longestPalindrome(string s)

size = s.length

longestStart = 0

max = 1

bool table[size][size] = {false}

for (i = 0; i < size; i++)

table[i][i] = true

for (i = 0; i < size-1; i++)

if s[i] == s[i+1]

table[i][i+1] = true

max = 2

longestStart = i

for (len = 3; len <= size; len++)

for (i = 0; i <= size - len; i++)

j = size - len + 1

if s[i] == s[j] && table[i+1][j-1] == true

max = len

longestStart = i

table[i][j] = true

return max