CS 325: PROJECT 2

GROUP 29: JACOB MASTEL, ROBERT ERICK, CERA OLSON

Date: 10 May 2015.

Contents

1. Changeslow Algorithm

The changeslow algorithm can also be called brute force or divide and conquer. We used the algorithm to find the minimum number of coins to make i and $K - i$ cents.

1.1. Pseudocode

. This is the pseudocode for the Changeslow Algorithm. It completes the requirement for Question 2.

```
FUNCTION addTwo(first ,second):
    RETURN [[x+y for x,y in zip(first[0],second[0])],
                    first[1]+second[1]]
ENDFUNCTION

FUNCTION changeslow(array ,K):
    IF K in array:
        out=[0]*len(array)
        out[array.index(K)]=1
        RETURN [out ,1]
    ELSE:
        minimum=None
        for i in array:
            IF not i<K: break
            ENDIF
            result1=changeslow(array ,i)
            result2=changeslow(array ,K-i)
            result=addTwo(result1 ,result2)
            IF minimum==None OR result[1]<minimum[1]:
                minimum=result
            ENDIF
        ENDFOR
        RETURN minimum
    ENDIF
ENDFUNCTION
```

2. Greedy Algorithm

The greedy algorithm starts with the largest values and goes through the lower values ignored to determine the lowest number of coins to make i and $K - i$ cents.

2.1. Pseudocode

. This is the pseudocode for the Greedy Algorithm. It completes the requirement for Question 2.

```
FUNCTION addTwo(first,second):
    RETURN [[x+y for x,y in zip(first[0],second[0])],
                    first[1]+second[1]]
ENDFUNCTION

FUNCTION changegreedy(array,K,i=0):
    zeroarray=[0]*len(array)
    IF K==0:
        RETURN [zeroarray,0]
    ELSE:
        index=len(array)-1-i
        biggest=array[index]
        howmany=int(K/biggest)
        deduct=biggest*howmany
        zeroarray[index]=howmany
        resulthere=[zeroarray,howmany]
        resultbelow=changegreedy(array,K-deduct,i+1)
        RETURN addTwo(resulthere,resultbelow)
    ENDIF
ENDFUNCTION
```

3. Dynamic Programming

Dynamic programming stores the calculated values to reduce the running time for larger values.

3.1. Pseudocode

. This is the pseudocode for the Dynamic Programming. It completes the requirement for Question 2.

```
FUNCTION addTwo(first,second):
    RETURN [[x+y for x,y in zip(first[0],second[0])],
                    first[1]+second[1]]
ENDFUNCTION

FUNCTION changedp(array,K):
    table={}
    rows=[(i,j) for i,j in enumerate(array)]
              ENDFOR
    cols=[(i,j) for i,j in enumerate(range(K+1))]
              ENDFOR
    for row in rows:
```

```
        r,rv=row
        table[r]=[]
        for col in cols:
            c,cv=col
            IF r==0 AND c==0:
                frm=(0,0)
                table[r]+=[(0,frm)]
            ELSEIF r==0:#top row
                backval=table[r][c-rv][0]+1
                frm=(r,c-rv)
                table[r]+=[(backval,frm)]
            ELSEIF cv<rv:#beginning of a row
                rowabove=r-1
                cellaboveval=table[rowabove][c][0]
                frm=(r-1,c)
                table[r]+=[(cellaboveval,frm)]
            else:#last part of a row
                rowabove=r-1
                cellaboveval=table[rowabove][c][0]
                backval=table[r][c-rv][0]+1
                IF cellaboveval<backval:
                    frm=(r-1,c)
                    table[r]+=[(cellaboveval,frm)]
                ELSE:
                    frm=(r,c-rv)
                    table[r]+=[(backval,frm)]
            ENDIF
                ENDIF
ENDFOR
    ENDFOR
out=[0]*len(array)
r=len(table)-1
c=len(table[r])-1
mincoins=table[r][c][0]
current=(r,c)
pcol=current[1]
while True:
    r=current[0]
    c=current[1]
    rv=rows[r][1]
```

```
        current=table[r][c][1]#next
        ccol=current[1]
        IF ccol!=pcol:
            out[array.index(rv)]+=1
        ENDIF
        pcol=ccol
        IF c==0:break
        ENDIF
    ENDWHILE
    RETURN [out,mincoins]
ENDFUNCTION
```

4. Questions

4.1. **Describe, in words, how you fill in the dynamic programming table in changedp. Justify why is this a valid way to fill the table?**

The methodology for filling the dynamic programming table is as follows:

First a table is constructed with columns [0..value we're looking for], interval of 1. The rows are [0..number of coins we have]. Let the indexes be r and c so that a each cell has coordinate (r, c). Let each row have a value of rv associated with the value for each coin, in increasing order. Let each column have a value cv associated with the values for each column, [0..value we're looking for]. c and cv have the same range.

If it is the first cell of the table, (0,0), the value is 0:

Otherwise, for the cells in the first row, the value of each cell c1 is the value of the cell c2 which occurs rv intervals prior, plus 1 (where rv is the row's value). Example: if row 0 represents a 1 cent coin, its "row value" is 1. Therefore, for cell (0,1), it's value will be the value of cell to its left by 1 (rv), which is cell (0,0), which has a value of 0, and this is increased by 1.

Otherwise, for cells in the lower rows, at the beginning of the row: all cells from (0,r) (where are is the row) inward are carried down from the cell immediately above each. Thus, which cv¡rv for each cell, carry down from above.

Otherwise, for cells in the endings of each row: compute the minimum of the value immediately above with the value compared with the value of the cell to the left rv hops, plus 1. That is, the minimum of a) value of cell(r-1,c) or b) (value of cell$(r, c - rv)) + 1$.

This methodology is correct. Firstly, smaller problems are being solved starting with the smallest rows and starting with the smallest values within those rows, working from the top left corner of the table down to the bottom right corner. The process builds from smallest solutions to greatest.

Secondly, populating the top row gives solutions as if the 1 coin were the only coin available. Each solution on the left helps develop the answer to its right. However, the subsequent rows are all populated thusly: the first portion of the row has solutions too small for that row's coin to solve, so the optimal solution from above is copied down. The second portion of the row is populated by using the matching optimal solution to the left plus that row's coin, or the optimal solution above. This process allows new information to be introduced (the new, higher valued coin) and used in an optimal solution, or to use the solution above if it is superior. Each successive row can, potentially, copy down the optimal solutions developed by the lower valued coins.

In this way, there is an ongoing evaluation as to whether to accept the results from prior stages in the problem or use newly computed results.

4.2. **Give pseudocode for each algorithm.**
See Sections 1-3.

4.3. **Prove that the dynamic programming approach is correct by induction. That is, prove that $T[v] = min_{v(i)v} T[v - V[i]] + 1, T[0] = 0$ is the minimum number of coins possible to make change for value v.**

4.4. **Suppose V = [1, 5, 10, 25, 50]. For each integer value of A in [2010, 2015, 2020, ..., 2200] determine the number of coins that changegreedy and changedp requires. You can attempt to run changeslow however if it takes too long you can select smaller values of A and also run the other algorithms on the values. Plot the number of coins as a function of A for each algorithm. How do the approaches compare? ??**
??

4.5. **Suppose V_1 = [1, 2, 6, 12, 24, 48, 60] and V_2 = [1, 6, 13, 37, 150]. For each integer value of A in [2000, 2001, 2002, ..., 2200] determine the number of coins that changegreedy and changedp requires. If your algorithms run too fast try [10,000, 10,001, 10,003, ..., 10,100]. You can attempt to run changeslow however if it takes too long you can select smaller values of A and also run all three algorithms on the values. Plot the number of coins as a function of A for each algorithm. How do the approaches compare? ??**

??

??

??

4.6. **Suppose V = [1, 2, 4, 6, 8, 10, 12, ..., 30]. For each integer value of A in [2000, 2001, 2002, ..., 2200] determine the number of coins that changegreedy and changedp requires. You can attempt to run changeslow however if it takes too long you can select smaller values of A and also run all three algorithms on the values. Plot the number of coins as a function of A for each algorithm. ??**

??

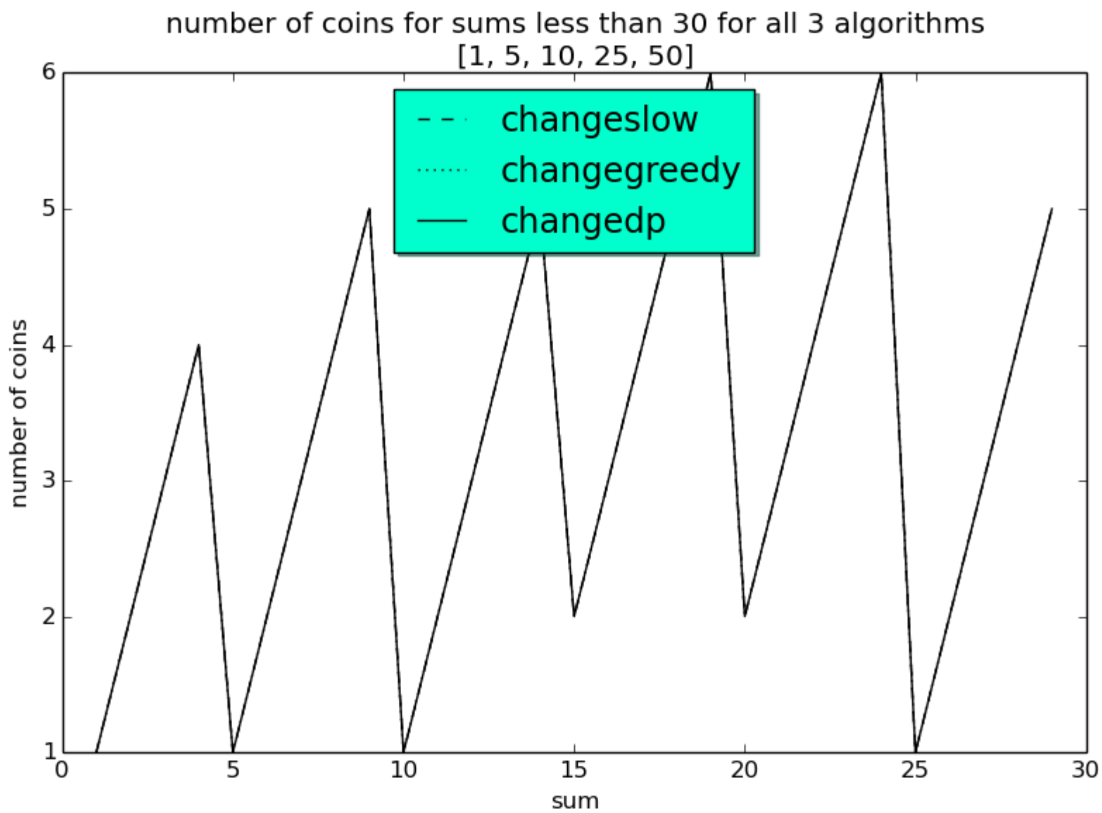


FIGURE 1. Changeslow Algorithm

4.7. **For the above situations, determine (experimentally) the running times of the algorithms by fitting trend lines to the data or analyzing the log-log plot. Graph the running time as a function of A. Compare the running times of the different algorithms.**

4.8. **Use the data from questions 4-6 and any new data you have generated. Plot running times as a function of number of denominations (i.e. V=[1, 10, 25, 50] has four different denominations so n=4). Does the size of n influence the running times of any of the algorithms?**

4.9. **Suppose you are living in a country where coins have values that are powers of p, $V = [p^0, p^1, p^2, , p^n]$. How do you think the dynamic programming and greedy approaches would compare? Explain.**

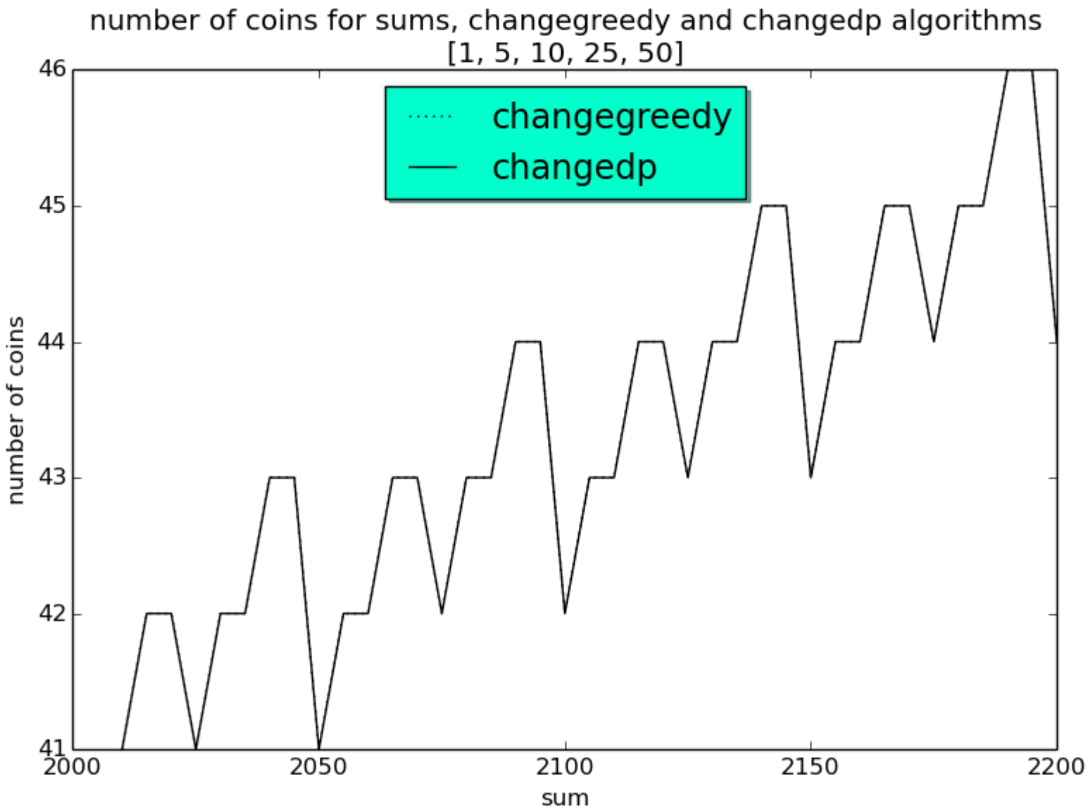


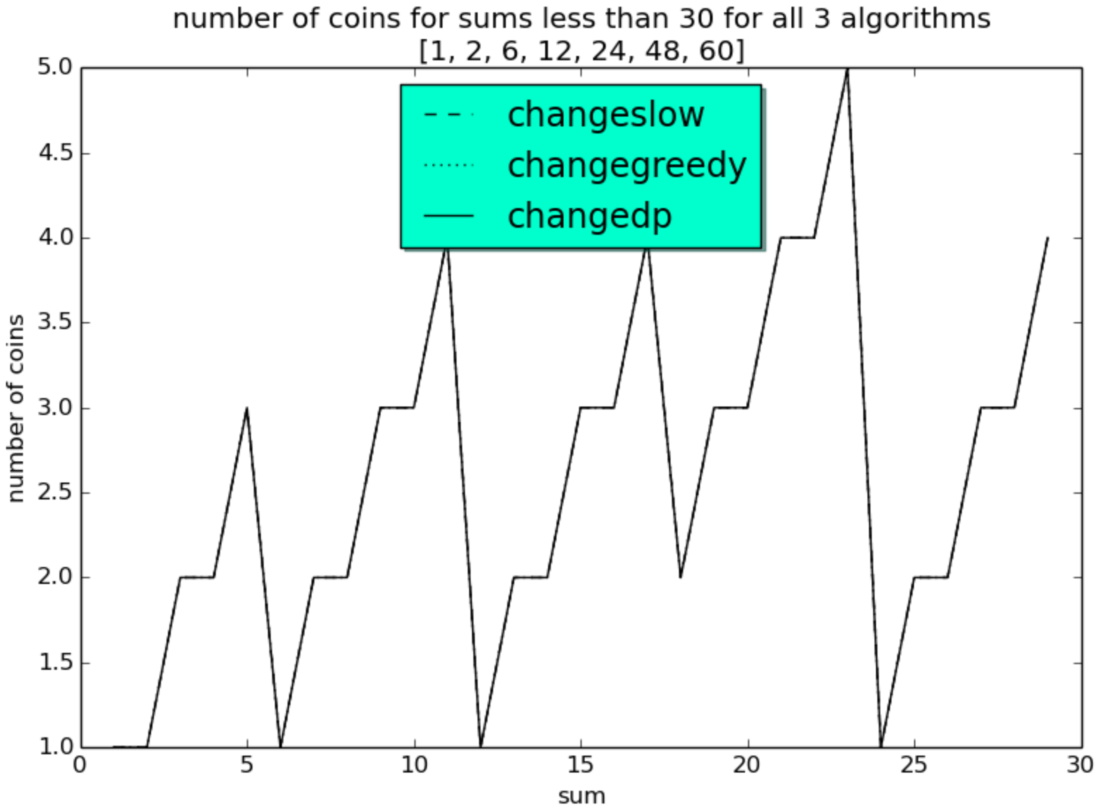
Figure 2. Changeslow Algorithm

5. Appendices

5.1. Code

. The main used to determine the results for Project 2.

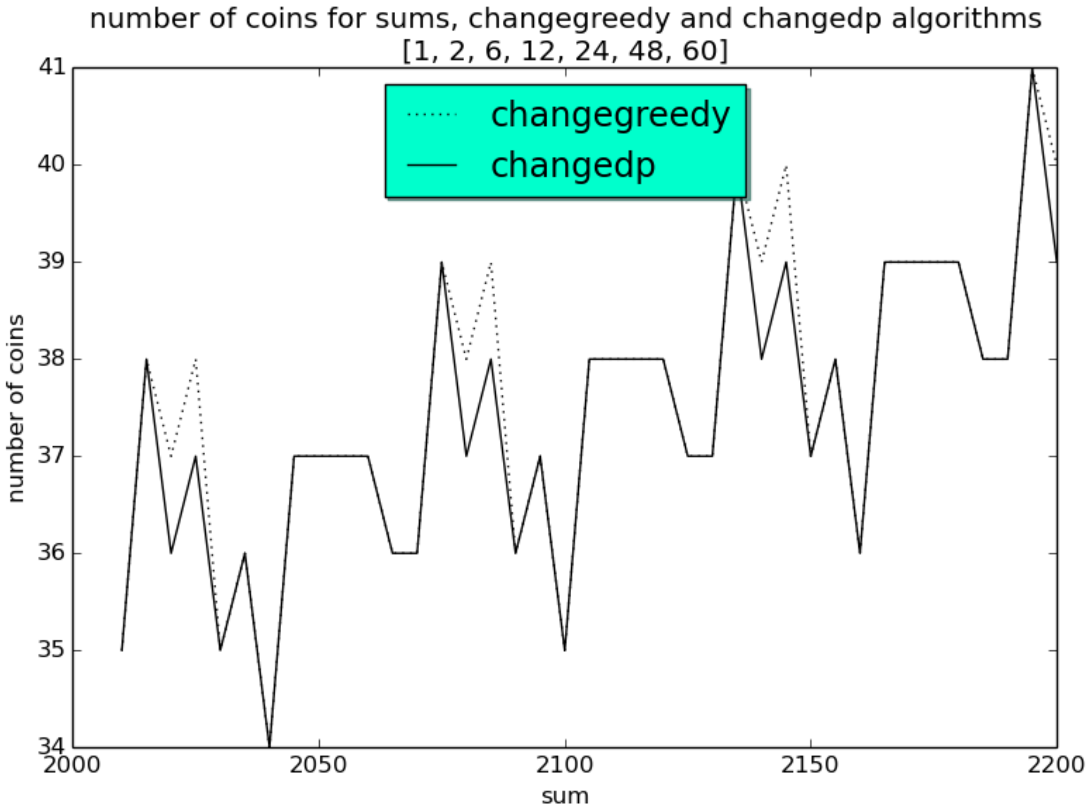
```
import os
import sys
from project2_rbt import *

def convert_to_array(V):
        V = V.replace("[", "")
        V = V.replace("]", "")
        V = V.replace(" ", "")
        V = V.replace("\n", "")
        V = V.split(',')
```



[h!]

FIGURE 3. Greedy Algorithm

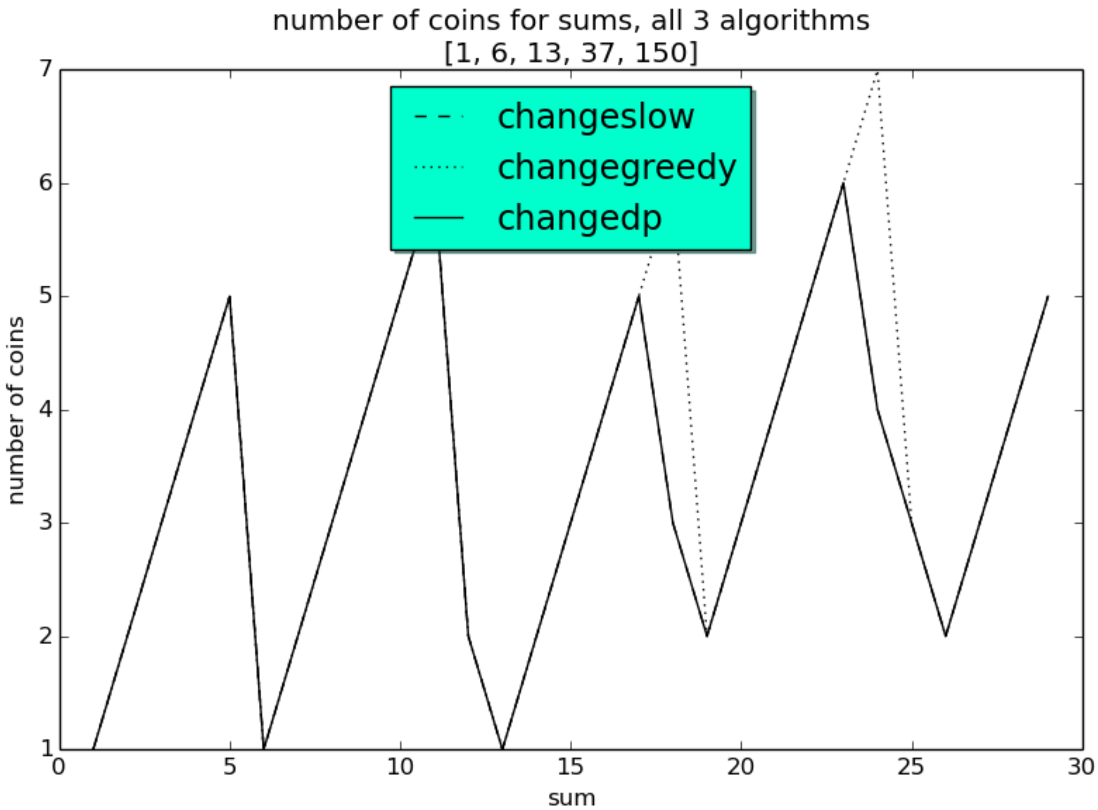
```
    return [int(v) for v in V]


def process_file(fname):
    f = open(fname, 'r')

    save_name = fname.replace(".txt", "")
    w = open(save_name+'change.txt', 'w')

    while True:
        V = f.readline()
        A = f.readline()
```



[h!]

FIGURE 4. Greedy Algorithm

```
                if not A: # EOF
                        break

                A = int(A)
                V = convert_to_array(V)

                res = changedp(V,A)
                w.write('{0}\n{1}\n'.format(res[0], res[1]))

if __name__ == "__main__":
        if len(sys.argv) != 2:
                print "Argument error"
```



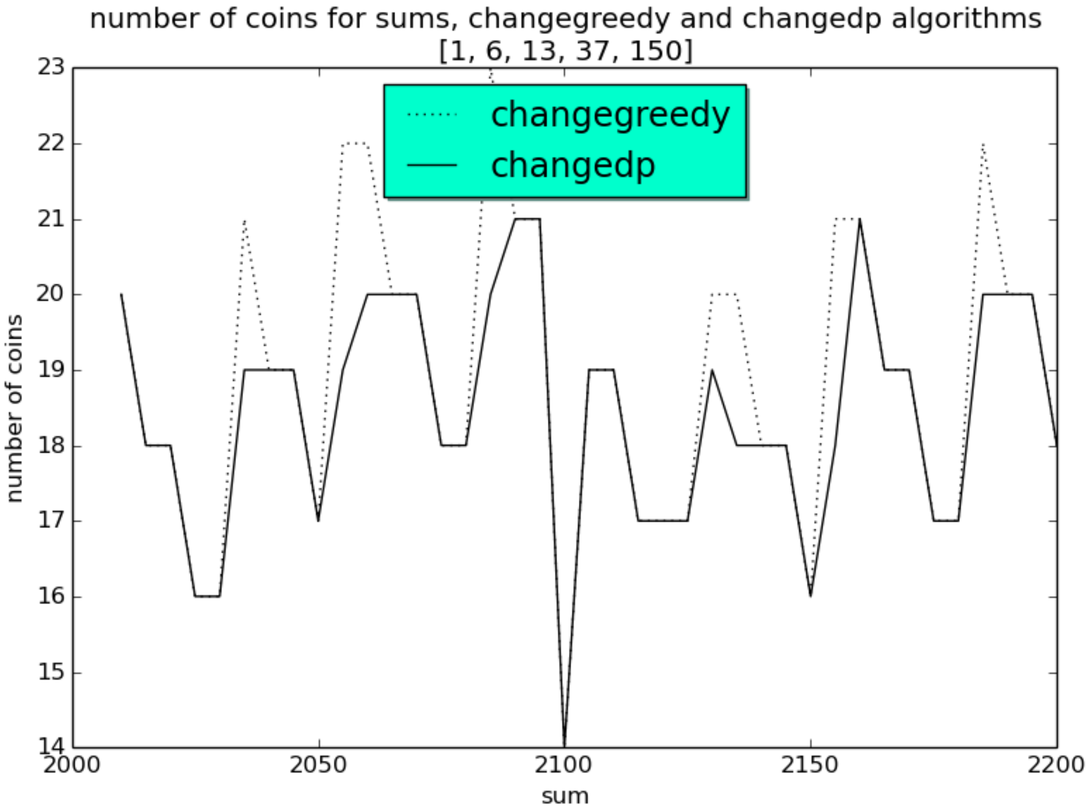
[h!]

FIGURE 5. Greedy Algorithm

```
    if os.path.isfile(sys.argv[1]):
            process_file(sys.argv[1])
    else:
            print 'File,␣{0},␣does␣not␣exist'.format(sys.argv[1])
```

5.2. **Tests**
. This contains the conditions and tests for our algorithms.

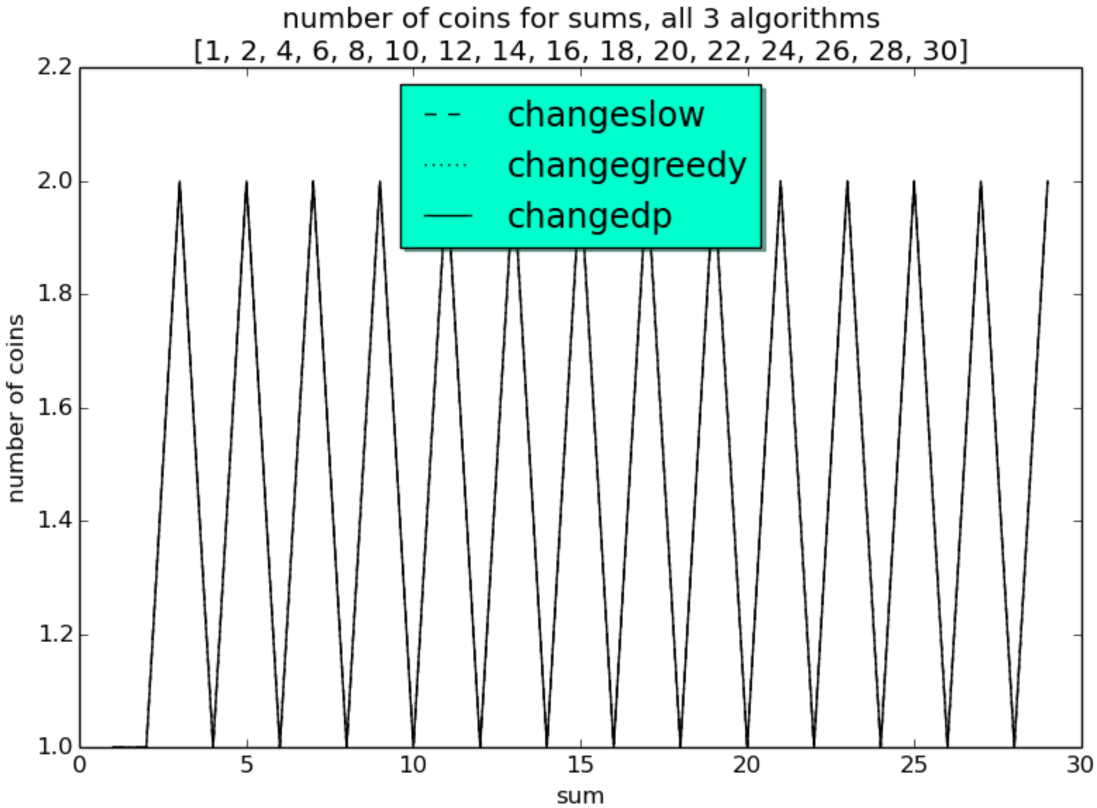
```
def addTwo(first,second):
    return [[x+y for x,y in zip(first[0],second[0])],
                first[1]+second[1]]

def changeslow(array,K):
```



[h!]

Figure 6. Greedy Algorithm

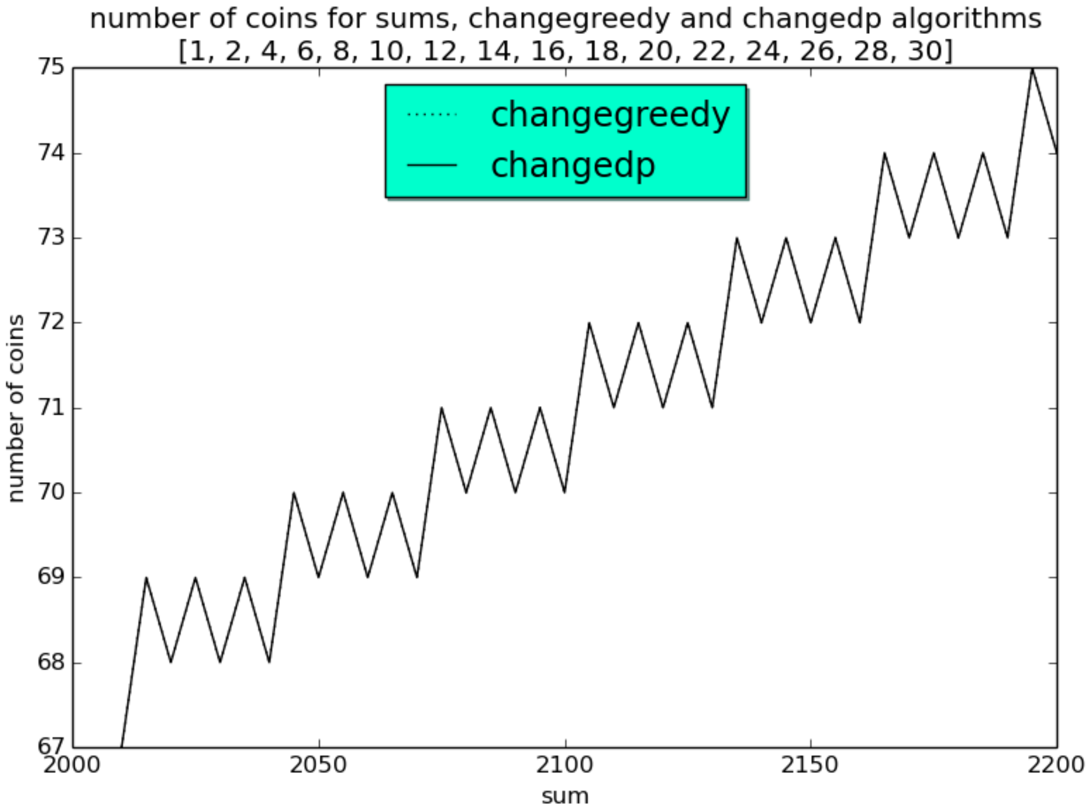
```
if K in array:
    out=[0]*len(array)
    out[array.index(K)]=1
    return [out,1]
else:
    minimum=None
    for i in array:
        if not i<K: break
        result1=changeslow(array,i)
        result2=changeslow(array,K-i)
        result=addTwo(result1,result2)
        if minimum==None or result[1]<minimum[1]:
            minimum=result
```



[h!]

FIGURE 7. Dynamic Programming

```
        return minimum

def changegreedy(array,K, i=0):
    zeroarray=[0]*len(array)
    if K==0:
        return [zeroarray,0]
    else:
        index=len(array)-1-i
        biggest=array[index]
        howmany=int(K/biggest)
        deduct=biggest*howmany
        zeroarray[index]=howmany
        resulthere=[zeroarray,howmany]
```



[h!]

Figure 8. Dynamic Programming

```
        resultbelow=changegreedy(array,K-deduct,i+1)
        return addTwo(resulthere,resultbelow)

def changedp(array,K):
    table={}
    rows=[(i,j) for i,j in enumerate(array)]
    cols=[(i,j) for i,j in enumerate(range(K+1))]

    for row in rows:
        r,rv=row
        table[r]=[]
        for col in cols:
            c,cv=col
```

```
        if r==0 and c==0:
            frm=(0,0)
            table[r]+=[(0,frm)]
        elif r==0:#top row
            backval=table[r][c-rv][0]+1
            frm=(r,c-rv)
            table[r]+=[(backval,frm)]
        elif cv<rv:#beginning of a row
            rowabove=r-1
            cellaboveval=table[rowabove][c][0]
            frm=(r-1,c)
            table[r]+=[(cellaboveval,frm)]
        else:#last part of a row
            rowabove=r-1
            cellaboveval=table[rowabove][c][0]
            backval=table[r][c-rv][0]+1
            if cellaboveval<backval:
                frm=(r-1,c)
                table[r]+=[(cellaboveval,frm)]
            else:
                frm=(r,c-rv)
                table[r]+=[(backval,frm)]

out=[0]*len(array)
r=len(table)-1
c=len(table[r])-1
mincoins=table[r][c][0]
current=(r,c)
pcol=current[1]
while True:
    r=current[0]
    c=current[1]
    rv=rows[r][1]

    current=table[r][c][1]#next
    ccol=current[1]
    if ccol!=pcol:
        out[array.index(rv)]+=1
    pcol=ccol
```

```
        if c==0:break
    return [out,mincoins]

if __name__=='__main__':
    # test 1
    # all should return [1,1,1,1]
    A = 15
    V = [1,2,4,8]
    print 'Test 1'
    print '  greedy: ', changegreedy(V,A)
    print '  slow:   ', changeslow(V,A)
    print '  dp:     ', changedp(V,A)

    # test 2
    # greedy should return [2,1,0,2]
    # slow and dp should return [0,1,2,1]
    A = 29
    V = [1,3,7,12]
    print 'Test 2'
    print '  greedy: ', changegreedy(V,A)
    print '  slow:   ', changeslow(V,A)
    print '  dp:     ', changedp(V,A)

    # test 3
    # all should return [0,0,1,2]
    A = 31
    V = [1,3,7,12]
    print 'Test 3'
    print '  greedy: ', changegreedy(V,A)
    print '  slow:   ', changeslow(V,A)
    print '  dp:     ', changedp(V,A)
```

5.3. **Questions**

. This contains code that helps to answer the questions above.

```
import sys
from timeit import Timer
from project2_rbt import *
from multiprocessing import Process

def q4():
```

```
    f = open('../questions/q4.csv', 'w')
    f.write('A,changegreedy,changedp\n')

    V = [1,5,10,25,50]
    r = {}
    for A in range(2010, 2201, 5):
        r['g'] = changegreedy(V,A)[1]
        r['d'] = changedp(V,A)[1]

        f.write('{0},{1},{2}\n'.format(A, r['g'], r['d']))

    f.close()

def q5():
    f = open('../questions/q5.csv', 'w')
    f.write('A,changegreedy V1,changegreedy V2,changedp V1, changedp V2\n')

    V1 = [1,2,6,12,24,48,60]
    V2 = [1,6,13,37,150]
    r = {}
    for A in range(2000, 2201, 1):
        r['g1'] = changegreedy(V1, A)[1]
        r['g2'] = changegreedy(V2, A)[1]
        r['d1'] = changedp(V1, A)[1]
        r['d2'] = changedp(V2, A)[1]

        f.write('{0},{1},{2},{3},{4}\n'.format(A,r['g1'],r['g2'],r['d1']

    f.close()

def q6():
    f = open('../questions/q6.csv', 'w')
    f.write('A,changegreedy,changedp\n')

    V = []
    V.append(1)
    for v in range(2, 31, 2):
        V.append(v)

    r = {}
```

```
    for A in range(2000, 2201, 5):
        r['g'] = changegreedy(V,A)[1]
        r['d'] = changedp(V,A)[1]

        f.write('{0},{1},{2}\n'.format(A, r['g'], r['d']))

    f.close()

def time_slow():
    f = open('../questions/time_slow.csv', 'w')
    f.write('A,Time\n')

    V = [1,5,10,25]
    for A in range(2, 100, 2):
        t = Timer(lambda: changeslow(V,A)).timeit(number=3)
        f.write('{0},{1}\n'.format(A,t))

        if t >= 20:
            break

    f.close()

def time_greedy():
    f = open('../questions/time_greedy.csv', 'w')
    f.write('A,Time\n')

    V = [1,5,10,25]
    for A in range(100, 1000001, 100):
        t = Timer(lambda: changegreedy(V,A)).timeit(number=3)
        f.write('{0},{1}\n'.format(A,t))

        if t >= 10:
            break

    f.close()

def time_dp():
    f = open('../questions/time_dp.csv', 'w')
    f.write('A,Time\n')
```

```
    V = [1,5,10,25]
    for A in range(100, 1000001, 1000):
        t = Timer(lambda: changedp(V,A)).timeit(number=3)
        f.write('{0},{1}\n'.format(A,t))

        if t >= 2:
            break

    f.close()

if __name__ == "__main__":
    print 'q4'
    q4()

    print 'q5'
    q5()

    print 'q6'
    q6()

    print 'slow'
    time_slow()

    print 'greedy'
    time_greedy()

    print 'dp'
    time_dp()
```