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.** Let T[v] be the minimum number of coins to create the sum v, where V[i] are the coin denominations.
Starting with T[0] = 0, when the sum is 0, is requires 0 coins.
There must exist a value $V[i] <= v$
The optimal solution then exists where $v - V[i]$
Taking T[0] = 0, T[v] = T[v - v[i]] + 1.
Because there are multiple possibilities that fit the scenarios above, T[v] must be the minimum solution of the possibilities.
Therefore, $T[v] = min_{V(i)v}T[v - V[i]] + 1$
**Based off code found at
$http : //www.ccs.neu.edu/home/jaa/CS7800.12F/Information/Handouts/dyn_prog.pdf$
written by Professor Aslam at Northeastern University for CS7800.

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?** See Figure 1.
This figure indicates the minimum number of coins required for sums less than 30. All threes algorithms find the same results, validating all the results. For small values we expected the number of coins to stay small, and all three algorithms valid this expectation. The algorithms all follow each other exactly.
See Figure 2.
The figure, only running the Greedy Algorithm and the Dynamic Programming algorithms, as the Changeslow algorithm takes too long at the larger sums. Both the Changegreedy and the Dynamic Programming Algorithms continue to find the same minimum number of coins for the sums through 2200. This indicates the accuracy of both algorithms. As expected, minimum number of coins continually makes an upward trending, increasing in the overall number of coins as the sum increases. While there is an up and down appearance to the graph, there is a distinct upward trend to the data.

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?** See Figure 3.
This figure found the same results for all the algorithms at the lower sum values (V_1). This, again, validates our results as correct and tells us that our algorithms are operating correctly. The graph appears as one line because all three algorithms return the same results.
See Figure 4.
Using V_1 again, we run the values using a larger sum value and the change greedy and the changed value return different results on some of the values in the higher range. This indicates an error in one of the algorithms that is causing an inaccurate count for the coins. The dynamic programming algorithm finds a smaller minimum number of coins for the higher values than those found by the greedy algorithm. Without a thorough evaluation of the algorithms, we cannot say whether it is the change greedy or the changed that is finding a few inaccurate counts.
See Figure 5.
This result proved to have a very interesting result. Using V_2, all three algorithms returned the same values initially. Between the sum 15 and 20, change greedy begins to returns values that are not the same as the other 2. This likely indicates an issue with out change greedy algorithm, especially when combined with the other results.
See Figure 6.
Using V_2 again, and the larger sum values, the change greedy and the changed charts return significantly different values through most of the domain. The changegreedy algorithm consistently returns values higher than that of the changdp algorithm. Following the other questions, the changeslow algorithm was removed from the graphs because of the lengthy run time of the algorithm.

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.** See Figure 7.
For the lower sums, all three algorithms are listed, and the results are consistent across all the algorithms. Because of the close values of V and the single coin that is valued at 1, the graph bounces between 1.0 and 2.0 coins for each sum. All the algorithms and the understanding of the question all remain consistent with this figure.
See Figure 8.
For the larger sums, the changeslow algorithm is dropped off, but the results are still

consistent. The trend remain in a steady upward trend with a jagged look to account for the differing coin values. Because the results remain consistent between the algorithms, our code appears to run correctly and efficiently.

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.** See Figure 9.
In these graphs we are comparing the running times between the algorithms. As is clear, the changeslow algorithm runs significantly slower than the other algorithms for sums greater than 15, except for at 25 where only 1 coin in required. The same dip in the running times would appear at sum = 50 but the overall trend in the running time would continue to increase well beyond the running time of the other two algorithms.
See Figure 10.
The log-log chart comparing changegreedy and changedp indicates that at the larger sum values, the change greedy algorithm runs more efficiently than the changedp algorithm. The difference is not significant on overall time - both algorithms stay below 1 second for sums under 100,000 - but when graphed in this form, it is clear that there is a distinct difference in the running times between the two algorithms. While there were some discrepancies in the results between these algorithms, the changegreedy algorithm is more efficient with its time.
See Figure 11.
This chart indicates the same as the other line of best fit. Changegreedy and changedp appear to have the same running time. Again, changeslow runs at such a slow speed in comparison to the other algorithms, it is not worth comparing at higher sum values.
See Figure 12.
The log-log plot for this situation indicates a much more consistent trend for changedp, although the end result is still the same: changegreedy is more efficient in the long run. If you draw a line of best fit through this plot, the running time for changegreedy does not increase as the sum goes up. This is contrary to changedp which has a continuing upward trend. For very large sums, changedp will become significantly less efficient than changegreedy.
See Figure 13.
Consistent with the other lines of best fit, the changeslow algorithm runs significantly slower than the other algorithms.
See Figure 14.
The log-log graph is a much more efficient comparison between the changegreedy and changedp algorithms. Because they are both so close running time, this is able to show the differences better, as with all the other log-log plots. This plot shows the same results as the other plots and the changegreedy is more efficient and the changedp algorithms will continue to increase in running time for very large sums.
See Figure 15.
Because of the wider range of coin values, the running times are much faster for this

algorithm. However, as expected, the changeslow algorithm runs much less efficiently when compared to the other two algorithms.
See Figure 16.
Again, consistent with previous log-log plots, the changegreedy proves to be the most efficient alorigthm. Changedp proves to increase its running time much faster than in the other algorithms. This indicates that the range of coin values does not effect the efficiency of the algorithm and as the sum increases, changedp becomes less efficient.

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?** See Figure 17.
For the lower sum (30), changeslow continues to be the most inefficient algorithm for the span of the number of coins, however, n does not seem to have a direct effect on the running time of the algorithm. For changegreedy and changedp though, there is a steady increase in the overall running time as the number of coins increases. For these two algorithms, the less choices in coin values, the more efficiently the algorithms will run.
See Figure 18.
For the larger sum (10,000), only the changegreedy and changedp algorithms are evaluated. Consistent with the results from the smaller sum, the algorithms have a steady upwards tread in running time as the number of coins available increases. However, changegreedy continues to be the more efficient algorithm.

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.** See Figure 19.
Based off the evaluations described above and Figure 19, the changegreedy would most definitely be a more efficient algorithm. The changedp would still be efficient enough to operate on a normal computer, but for vary large sums, the running time would get beyond what is practical. However, changegreedy does not appear to be increasing for this set of V. Changegreedy would continue to be the more efficient option for evaluation, the graphs and testing indicate no upward trend in the long run for very large sum values. *Note: Latex was causing errors with this final image. See the separate file to know the image to which we are referring. The conclusions drawn still remain.

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(',')

        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()

                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"
        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):
    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
        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]
        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()
```

5.4. **Figures.** Listed here are the figures, in order. Latex was not cooperating with putting the graphics into the report itself.

??
??
??
??
??
??
??
??
??
??
??
??
??
??
??
??
??
??

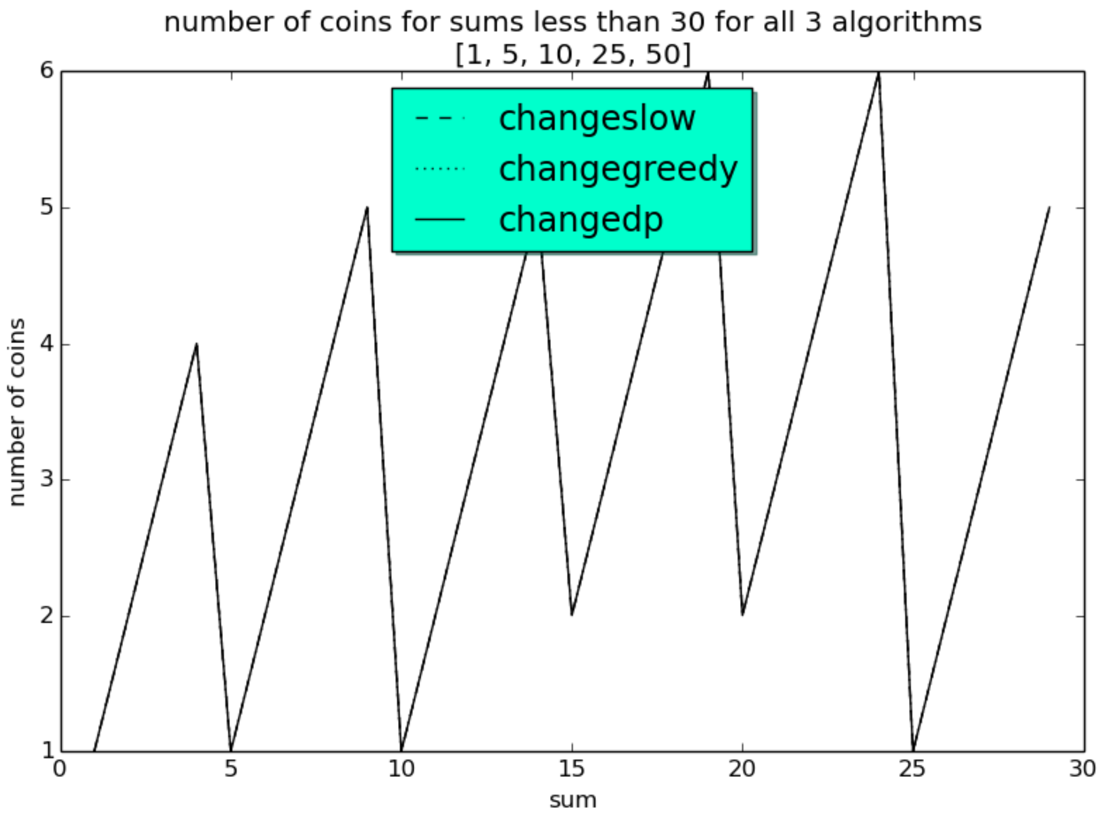

Figure 1. Number of Coins, $Sum > 30$, V = 1,5,10,25,50

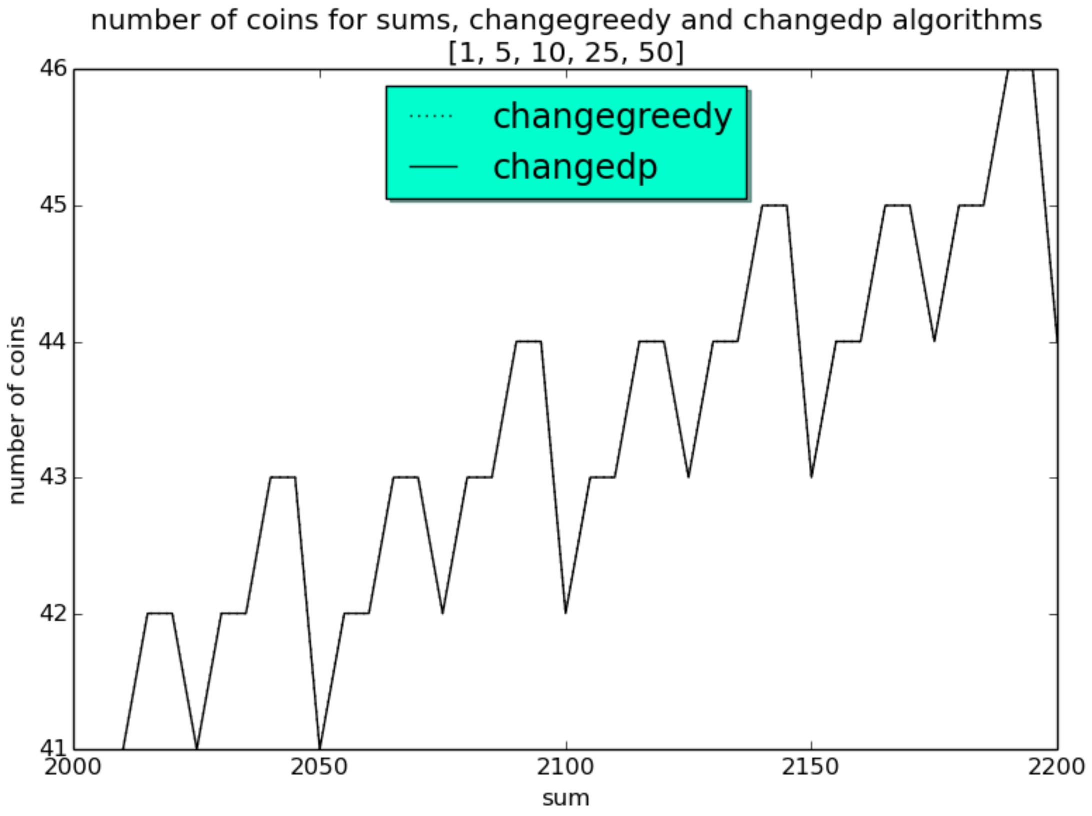

Figure 2. Number of Coins, $Sum > 2200$, V = 1,5,10,25,50

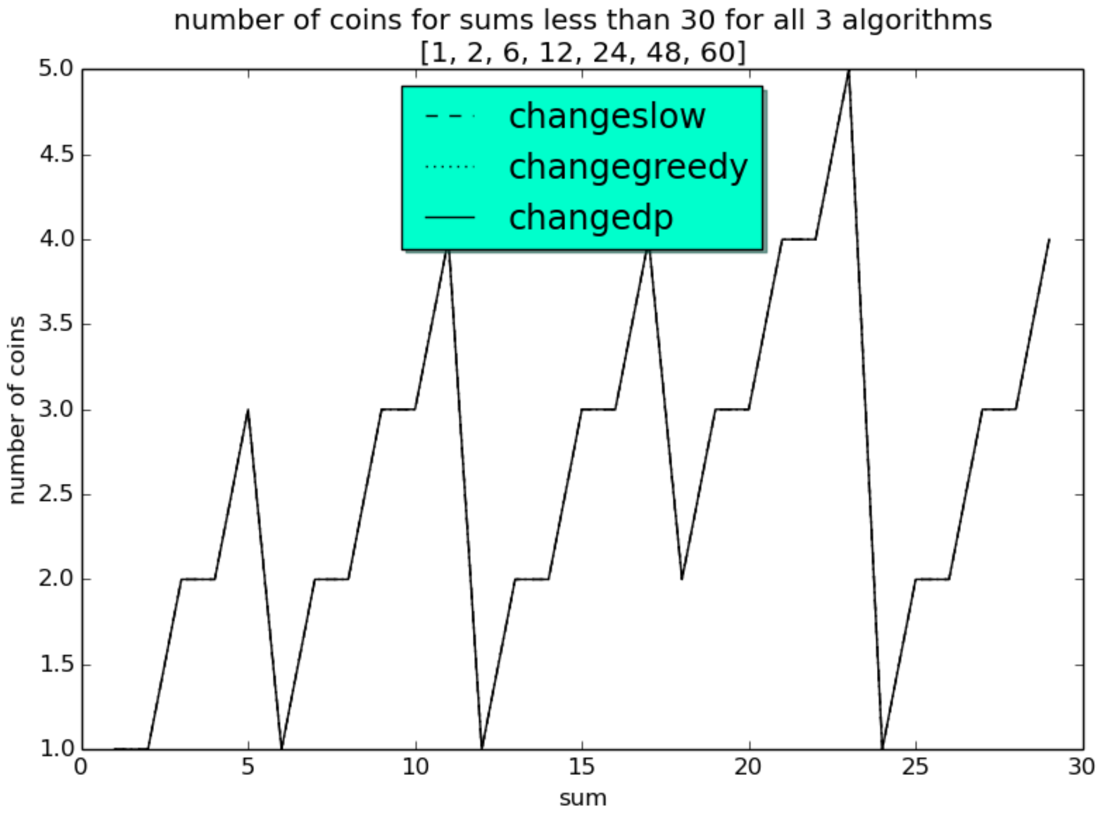

FIGURE 3. Number of Coins, $Sum > 30$, V = 1,2,6,12,24,48,60

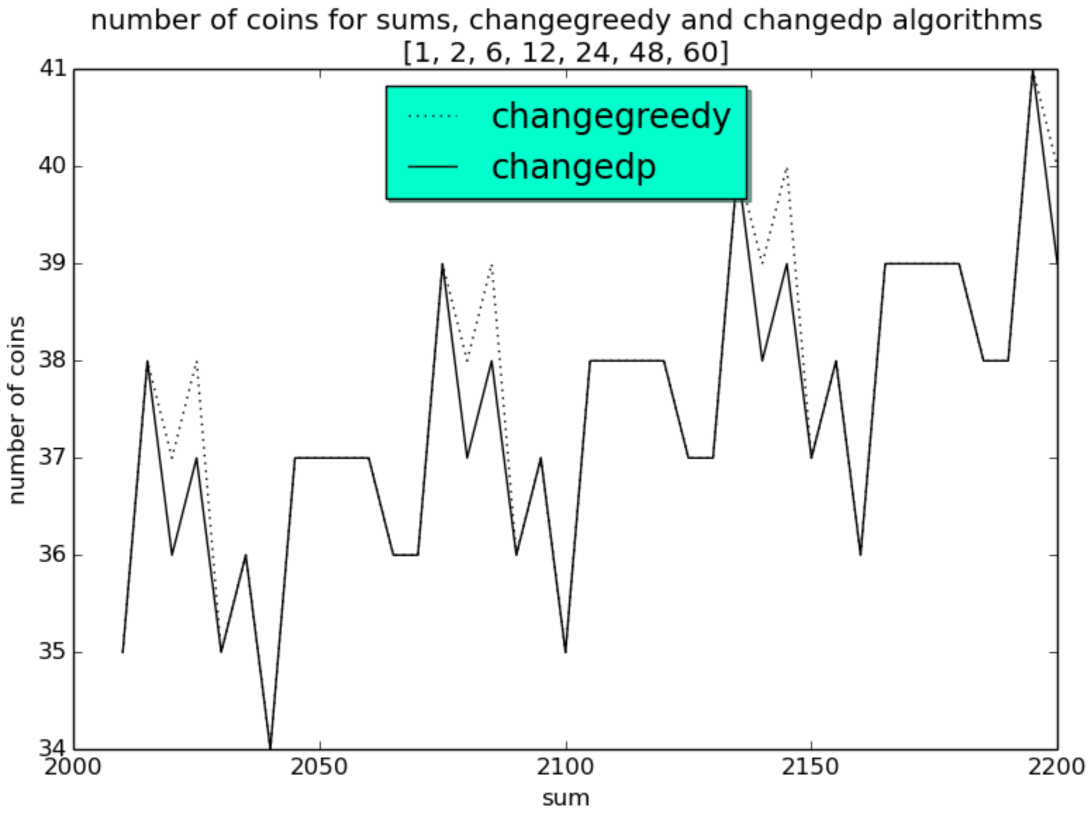

FIGURE 4. Number of Coins, $Sum > 2200$, V = 1,2,6,12,24,48,60

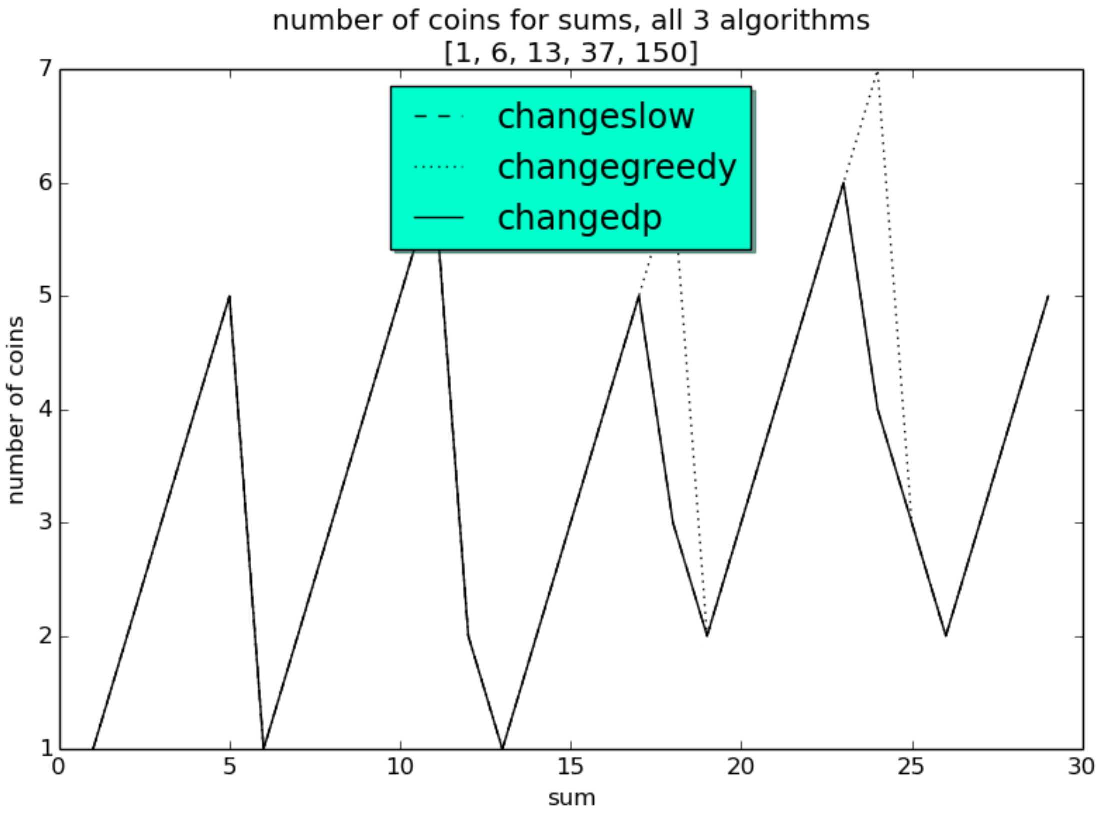

FIGURE 5. Number of Coins, $Sum > 30$, V = 1,6,13,37,150

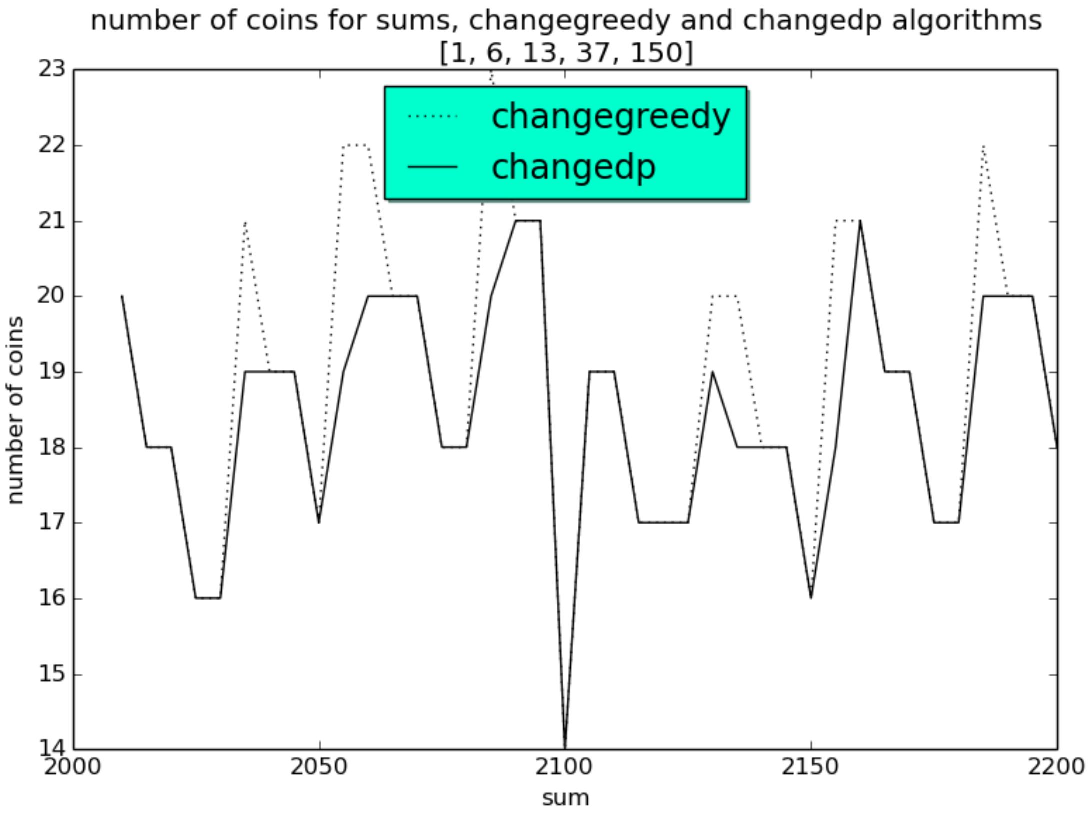

FIGURE 6. Number of Coins, $Sum > 2200$, V = 1,6,13,37,150

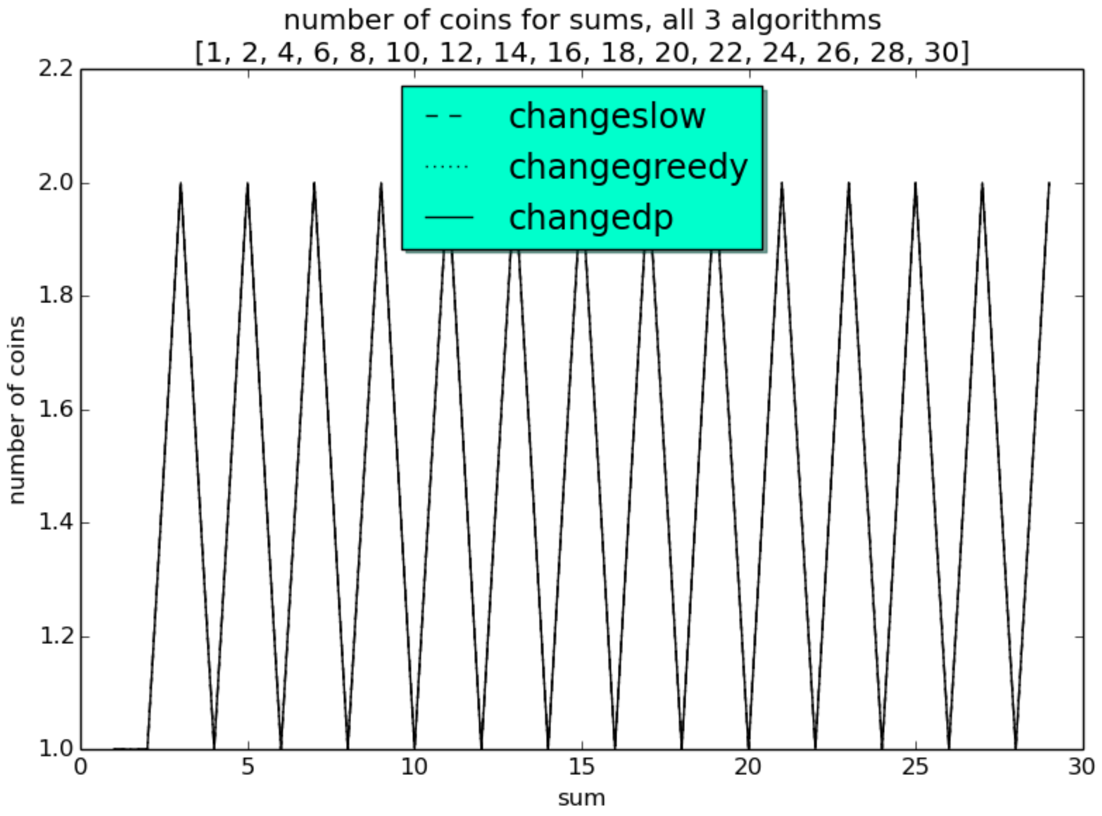

FIGURE 7. Number of Coins, $Sum > 30$, V = 1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30

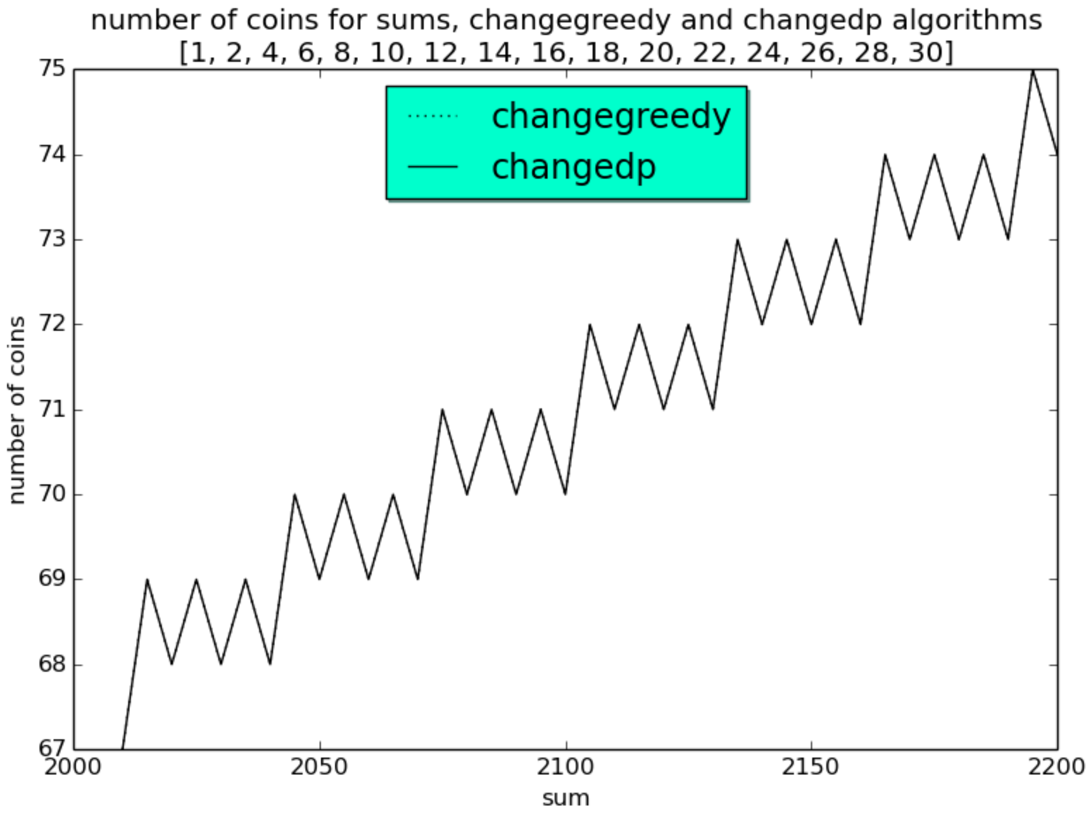

FIGURE 8. Number of Coins, $Sum > 2200$, V = 1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30

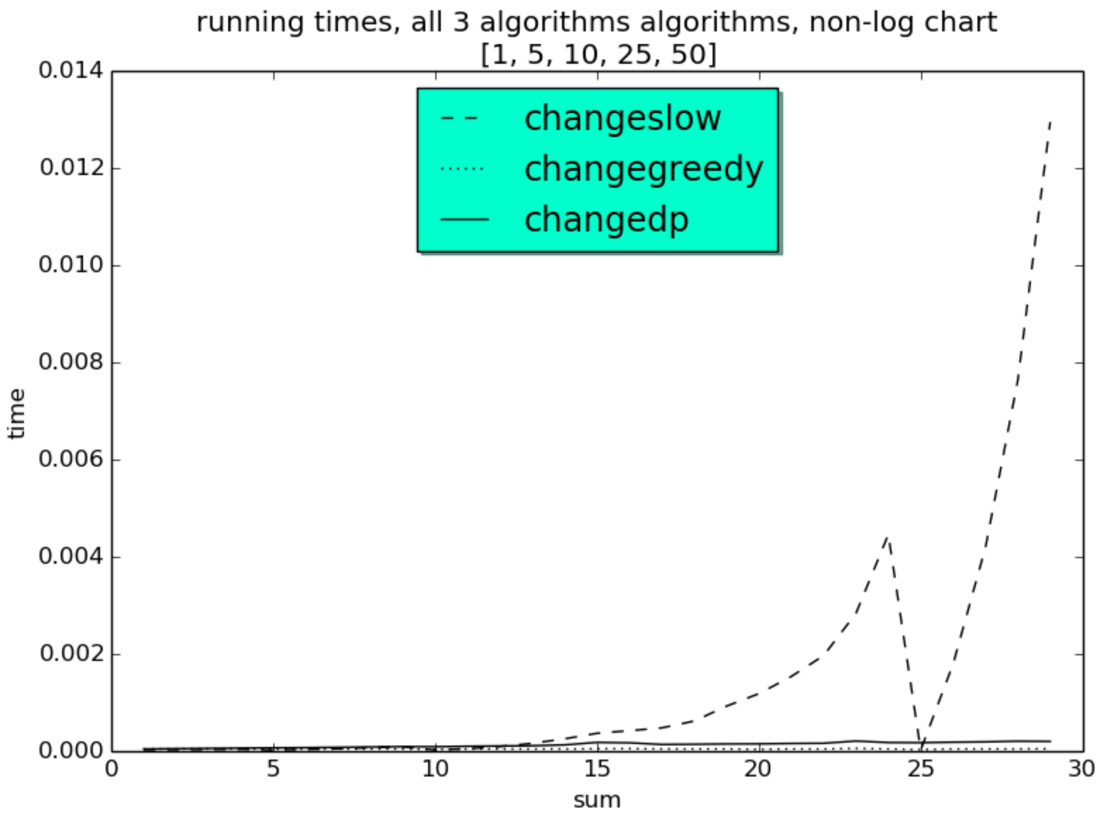

FIGURE 9. Line of Best Fit, V = 1,5,10,25,50

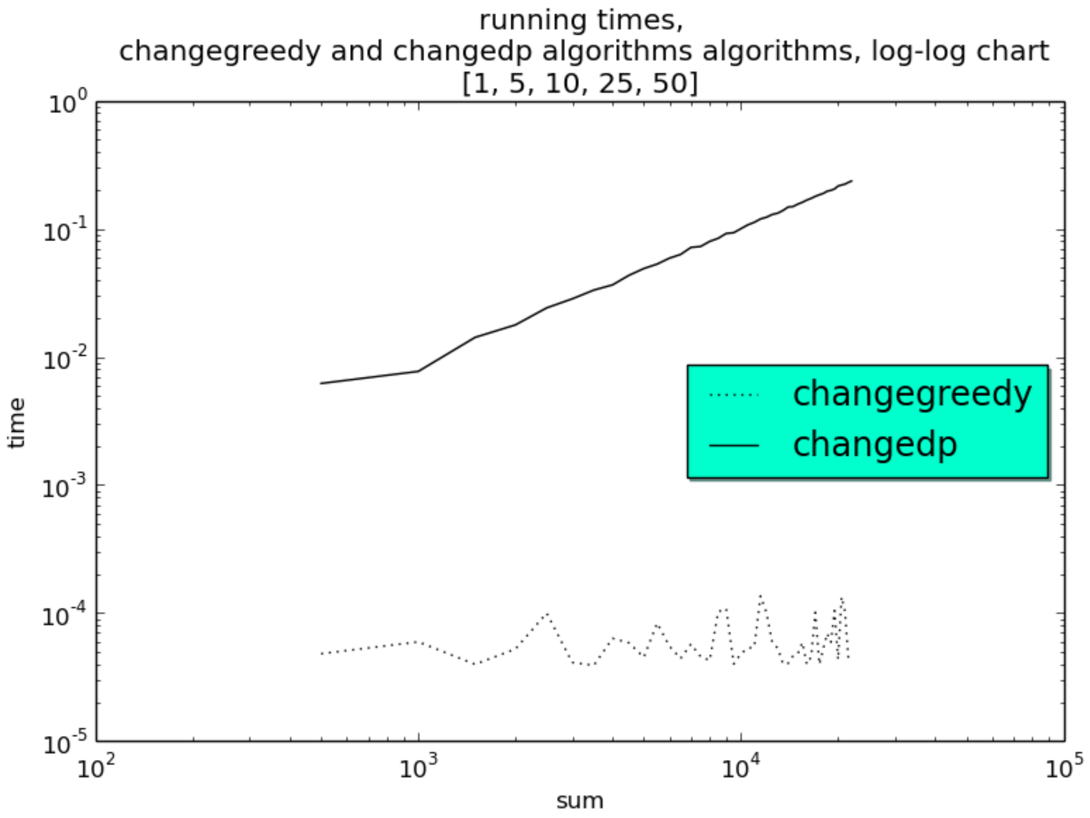

FIGURE 10. Log-Log, V = 1,5,10,25,50

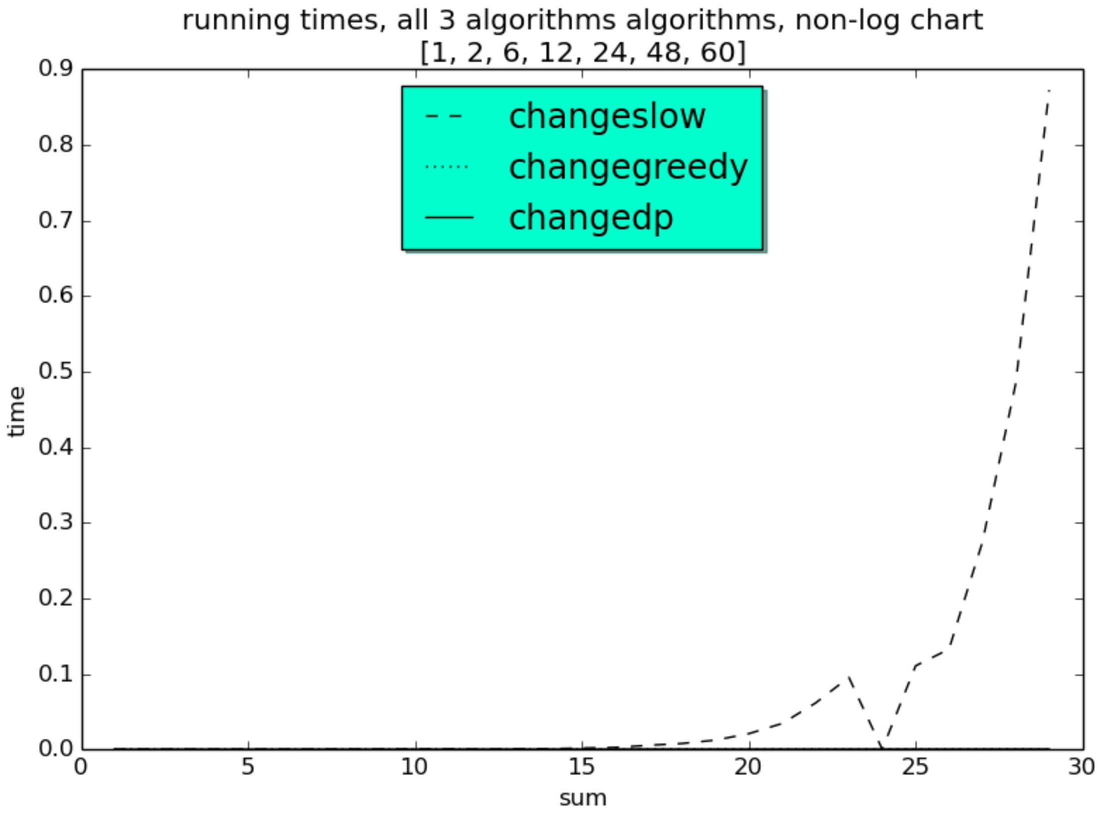

Figure 11. Line of Best Fit, V = 1,2,6,12,24,48,60

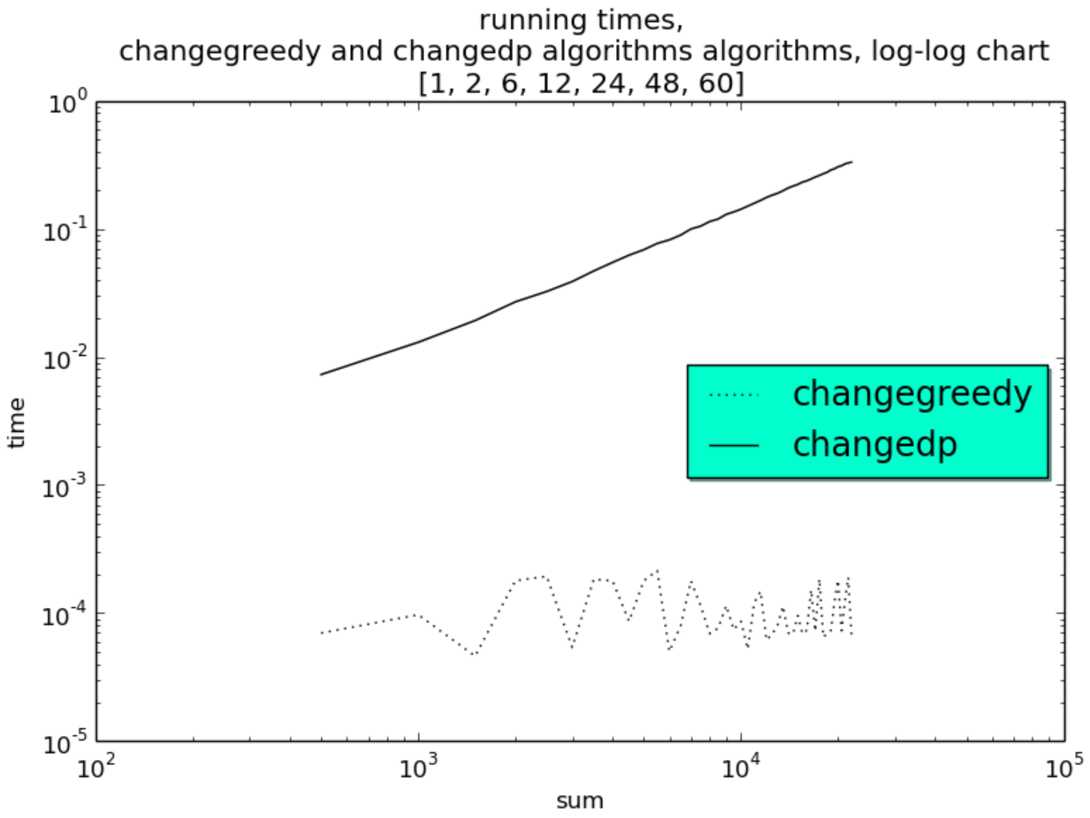

Figure 12. Log-Log, V = 1,2,6,12,24,48,60

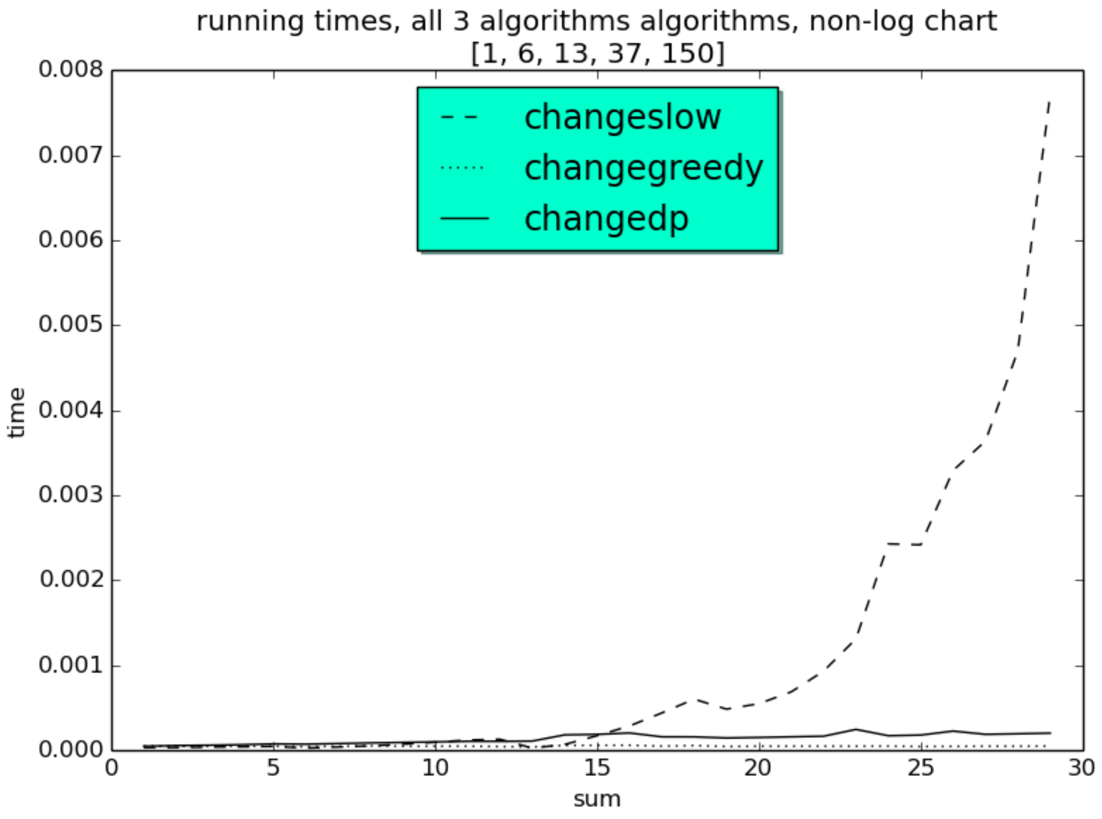

FIGURE 13. Line of Best Fit, V = 1,6,13,37,150

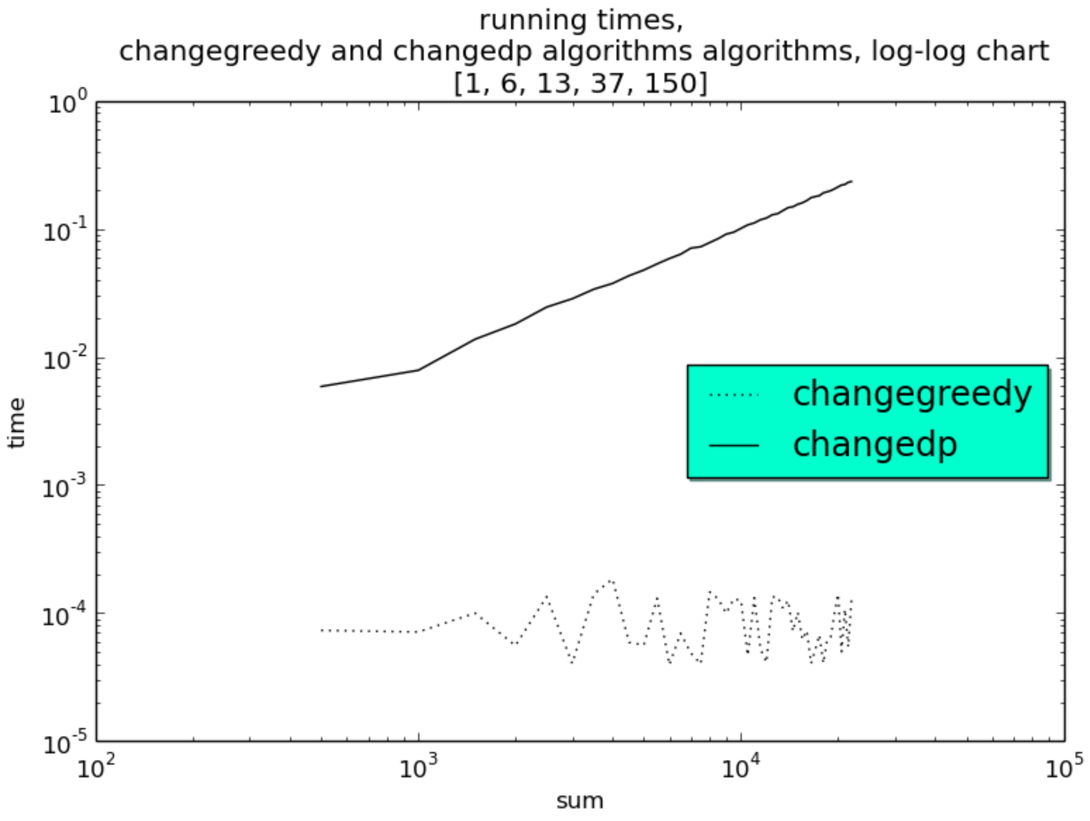

Figure 14. Log-Log, V = 1,6,13,37,150

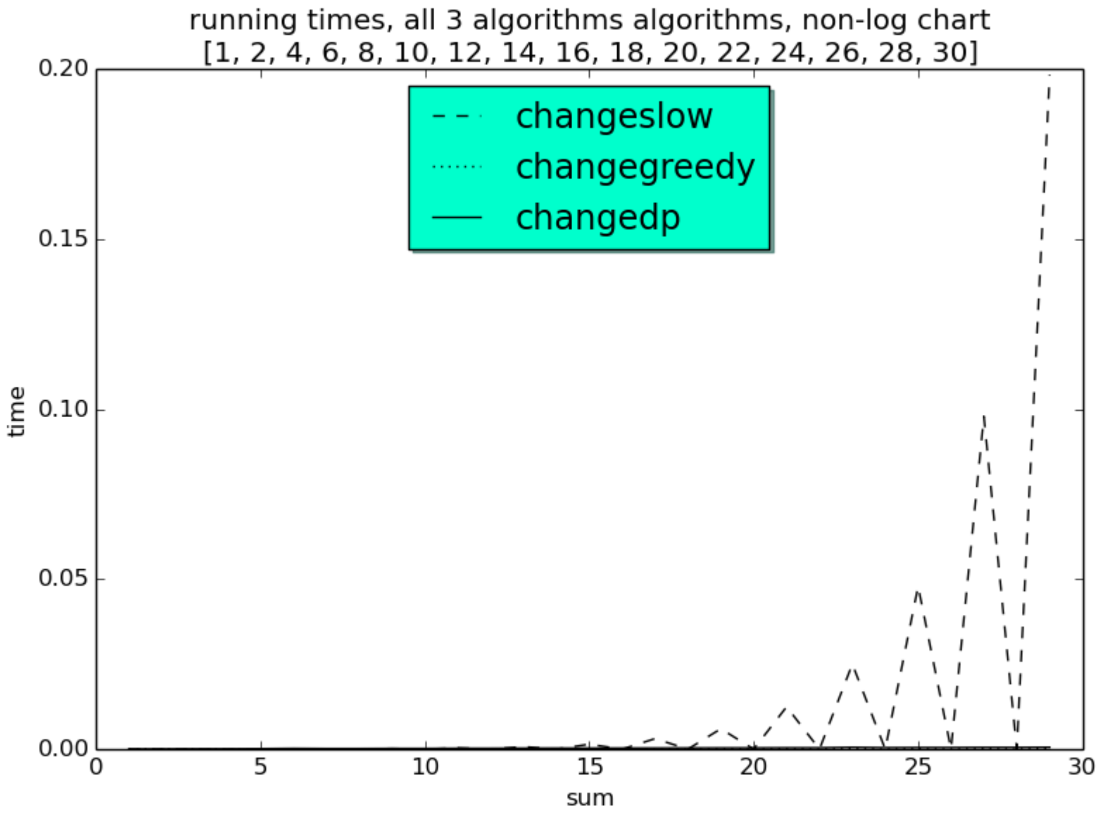

Figure 15. Line of Best Fit, V = 1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30

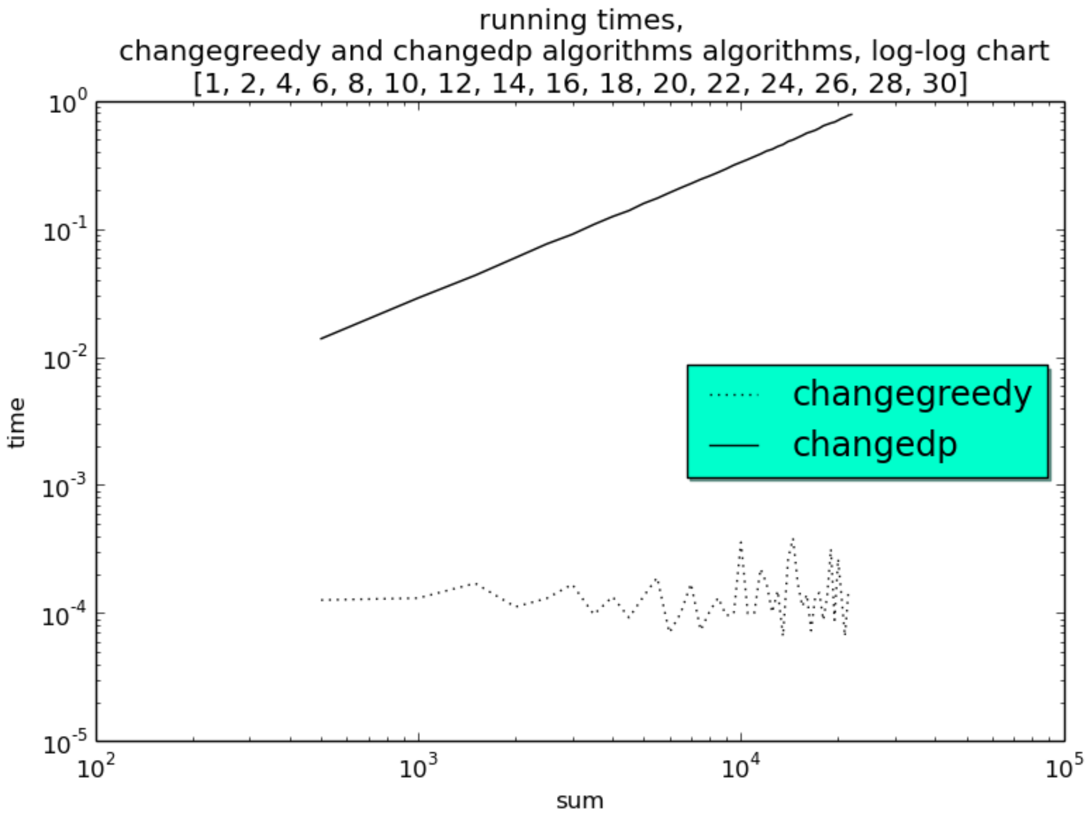

FIGURE 16. Log-Log, V = 1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30

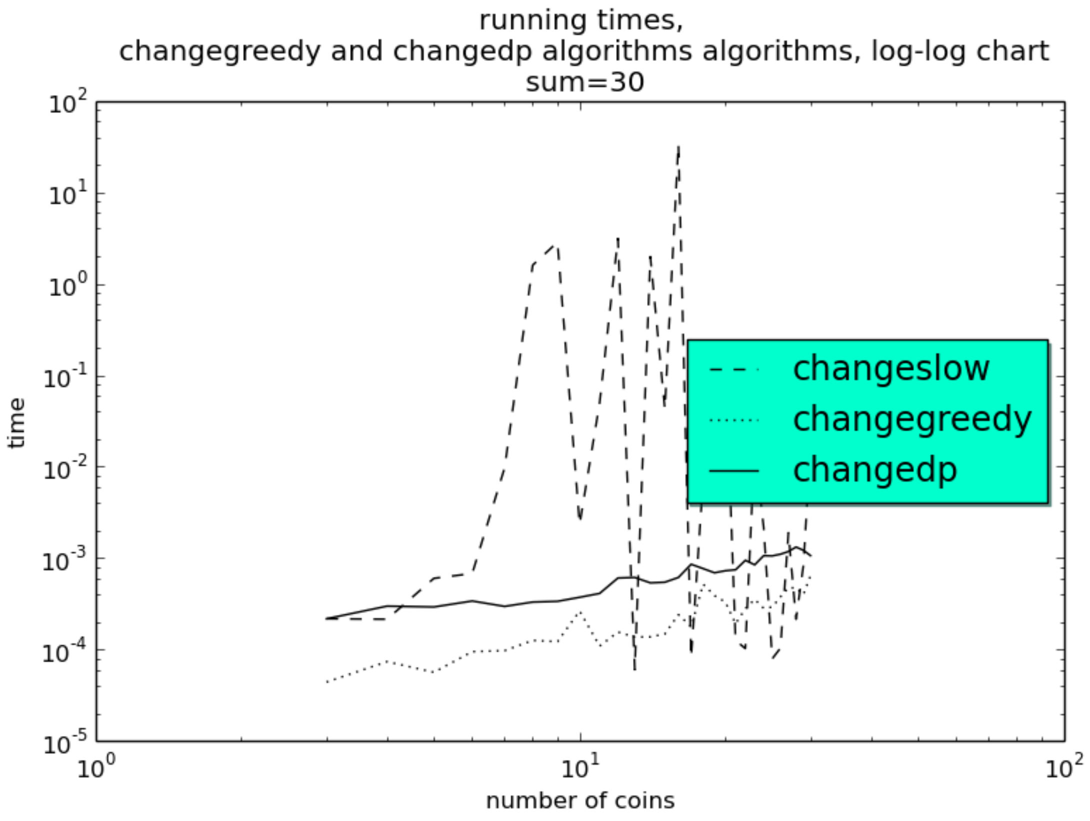

Figure 17. Log-Log, Sum = 30

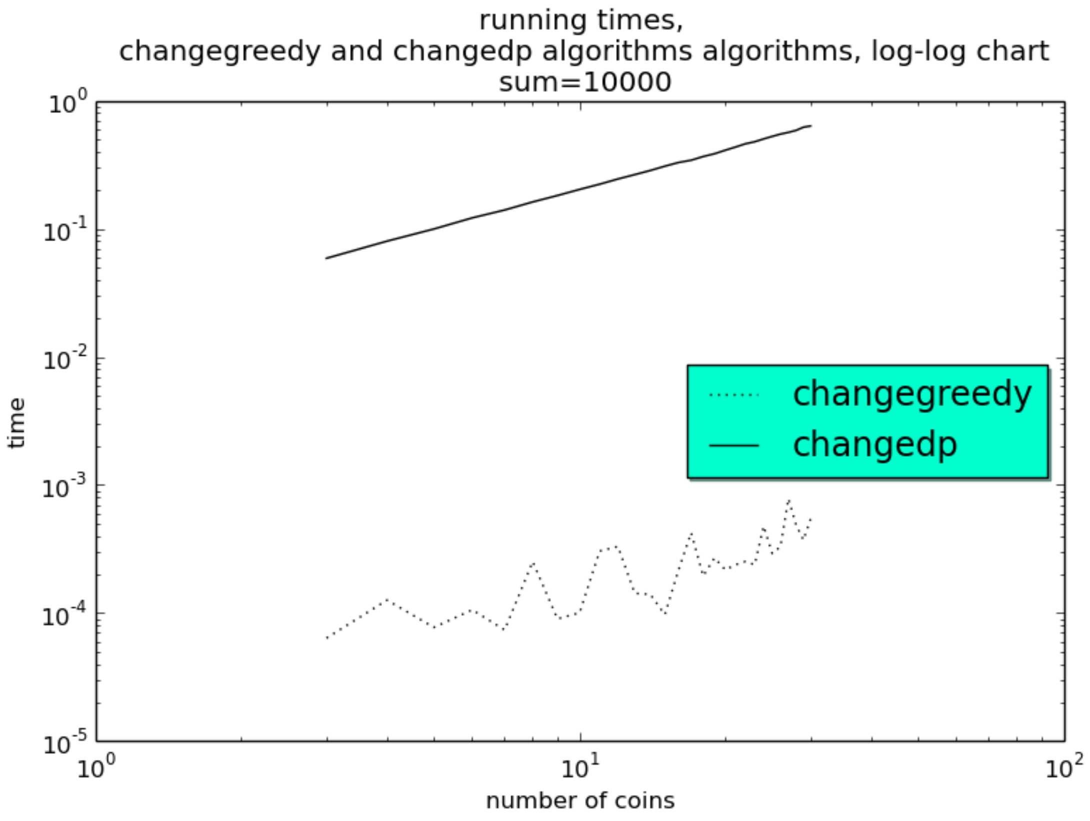

Figure 18. Log-Log, Sum = 10,000