COMP 555 PROBLEM SET 2

ELLIOTT HAUSER

1

1a. I can find only one condition that is necessary and sufficient to ensure that a solution to the change problem exists:

$$\exists c_i = 1$$

This states that one denomination c_i always be equal to one.[1] This is both sufficient because, for any integer M, M mod c_i = 0 when either condition holds, meaning that some combination of coins will always exist that adds up to M with no remainder. It is necessary because in the absence of $c_i = 1$, $\exists M \mod c_i \neq 0$.

1b. and 1c. The BetterChange algorithm in chapter 2 is given as:
BetterChange(M,**c**,d)

```
r ← M
for k ← 1 to d
    i_k ← r/c_k
    r ← r − c_k * i_k
return (i_1, i_2, ⋯ , i_d)
```

The approximation ratio of this algorithm is defined as $\max\limits_{|\pi|=n} \frac{\mathcal{A}(\pi)}{OPT(\pi)}$, where $\mathcal{A}(\pi) =$ BetterChange(M,**c**,d). In other words, this is the algorithm's worst performance compared to the optimal solution. To start, we'll specify this ratio for $0 \leq M \leq 100$. Since **c** and d are held constant, we'll only use M for π. The values of M for which BetterChange is incorrect are shown here:

Date: October 11, 2012.

[1]Another condition, $\exists c_i = M$ is sufficient but not necessary. This is also a bit of a cheat and will in practice rarely occur (i.e. the denomination of money is not alterable, and there is d/M probability that this condition would be satisfied by a random set of denominations, assuming M≥max(c))

$$\frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{\text{BetterChange}(M)}{OPT(M)}$$

$$0 \leq M \leq 100$$

$$\vdots$$

$$\text{for } M = 40, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{3}{2}$$

$$\text{for } M = 41, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{4}{3}$$

$$\text{for } M = 42, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{5}{4}$$

$$\text{for } M = 43, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{6}{5}$$

$$\text{for } M = 44, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{7}{6}$$

$$\vdots$$

$$\text{for } M = 65, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{4}{3}$$

$$\text{for } M = 66, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{5}{4}$$

$$\text{for } M = 67, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{6}{5}$$

$$\text{for } M = 68, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{7}{6}$$

$$\text{for } M = 69, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{8}{7}$$

$$\vdots$$

$$\text{for } M = 90, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{5}{4}$$

$$\text{for } M = 91, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{6}{5}$$

$$\text{for } M = 92, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{7}{6}$$

$$\text{for } M = 93, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{8}{7}$$

$$\text{for } M = 94, \frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{9}{8}$$

$$\vdots$$

So the approximation ratio for BetterChange where $0 \leq M \leq 100$ is max $\begin{cases} \frac{3}{2} \\ \frac{4}{3} \\ \frac{5}{4} \end{cases} = 1.5.$

Can this result be generalized for all $M \geq 0$?

There is a pattern to the above. In each of these cases, $M \geq 25$, and $15 \leq M \mod 25 < 20$. Also, $\frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{n+1}{n}$. I believe this is because $((((40\%25)\%20)\%10)\%5) = 3$ and $40\%20 = 2$; it is this portion of the formula that contributes the $+1$ to the numerator. For $15 \leq M \mod 25 < 20$ and $\frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{n+1}{n}$,

$$n = (M\%20) + (M\%25 - 15)$$

yielding:

$$\frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{n+1}{n} = \frac{(M\%20) + (M\%25 - 15) + 1}{(M\%20) + (M\%25 - 15)}$$

This is a long way of saying that, for c={1,5,10,20,25} the approximation ratio of BetterChange is 1.5 for any $M \geq 0$. the difference between numerator and denominator will only ever be 1, and this fraction will be greatest for the smallest possible value of n. Since $\frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{n+1}{n}$ only holds for $M \geq 25$ and$15 \leq M \mod 25 \leq 19$, $M = 40$ (i.e. $n = 2$) yields the maximum value of this function, 1.5.

1d. If we'd like to estimate the maximum value of $\frac{\mathcal{A}(\pi)}{OPT(\pi)}$ for any set of denominations c, we'll have to generalize the fraction $\frac{n+1}{n}$ and state it in terms of c.

For c = {1,5,10,20,25} and $M \geq 25$ and$15 \leq M \mod 25 \leq 19$,

$$\frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{n+i}{n} = \frac{2+1}{2} = 1.5$$

For another example $c = \{1, 20, 25\}$ and$M \geq 25$ and$15 \leq M \mod 25 < 20$, the same numbers hold. This suggests that the formula can be stated in terms of $c_{\max}$, $c_{\max -1}$, and c_1.

For $c = \{1, 15, 25\}$ and $M \geq 25$ and$5 \leq M \mod 25 \leq 14$,

$$\frac{n+i}{n} = \frac{2+4}{2} = 3.0$$

It is somewhat difficult to see the connection here, but we can induce a formula that explains these and other outcomes.

In all cases, n is the optimal number of coins. For the purposes of determining a maximum, we can assume that this is the lowest possible number where BetterChange will produce a discrepancy, $n = 2$, because this will maximize the fraction. Next we need to determine when BetterChange will be incorrect. This will be the case iff $\exists c_i >$

$\frac{c_{\max}}{2}$.We'll first consider the simplest case where there is one $c_i > \frac{c_{\max}}{2}$. In the two examples we've seen so far, $c_{\max} = 25$ and $c_i = \{15, 20\}$.

Next, we need to characterize the values of M that will be erroneously predicted. For c = {1,5,10,20,25} and $M \geq 25$ and$15 \leq M \mod 25 < 20$. How can we state these latter equations in terms of c?

$$M \geq 25 \equiv M \geq c_{\max}$$
$$15 \leq M \mod 25 < 20 \equiv \left(2 * c_{\max -1} - c_{max}\right) \leq M \mod c_{max} < c_{\max -1}$$

With these equations in hand, the one remaining problem is to determine the value of i. It's again difficult to see how, for $c = \{1, 5, 10, 20, 25\}$, $i = 1$, while for $c = \{1, 20, 25\}$, $i = 14$. But upon closer inspection, it is clear that

$$i = \text{BetterChange}\Big(((2 * c_{\max -1} - c_{max}), c, d) - 1$$

In other words, the difference between twice $c_{\max -1}$ and $c_{\max}$ is run through the BetterChange algorithm, and the result is decremented by one.

We should now test this admittedly complex system of equations with a random c, chosen for its difficulty. One feature of all of the cs evaluated so far is their common denominators. So, for this test, I'll choose a c comprised entirely out of primes.

$$c = \{1, 7, 11\}$$

Following the equations above, the errors in BetterChange should be given by

$$\left[\left(2 * c_{\max -1} - c_{max}\right) \leq M \mod c_{max} < c_{\max -1}\right] = \left[\left(2 * 7 - 11\right) \leq M \mod 11 < 7\right]$$
$$M_{\text{errors}} = \{14, 15, 16, 17, 25, 26, 27, 28 \cdots\}$$

So what is the approximation ratio in this case?

$$\frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{n+i}{n}$$
$$i = \text{BetterChange}\Big(((2 * c_{\max -1} - c_{max}), c, d) - 1$$
$$= \text{BetterChange}\Big((3, \{1, 7, 11\}, 3) - 1$$
$$= 2$$
$$\therefore$$
$$\frac{n+i}{n} = \frac{4}{2}$$

Is this correct? We can see that that for the first erroneous prediction, M=14, BetterChange would return one 11¢ coin and three 1¢ coins, whereas the optimal solution yields two 7¢ coins.[2]

So, to return to the question at hand, can we determine an upper bound on the approximation ratio? There is no limit. Because $\frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{n+i}{n}$, we know that the ratio will vary with i. As (erroneous) c get larger and larger, this ratio increases as well. For example, with $c = \{1, 75, 100\}$,

$$\frac{\mathcal{A}(\pi)}{OPT(\pi)} = \frac{n+i}{n}$$

$$\begin{aligned} i &= \text{BetterChange}\Big(((2 * c_{\max -1} - c_{max}), c, d) - 1 \\ &= \text{BetterChange}\Big((2 * 75 - 100, \{1, 75, 100\}, 3) - 1 \\ &= \text{BetterChange}\Big((50, \{1, 75, 100\}, 3) - 1 \\ &= 50 \\ &\therefore \end{aligned}$$

$$\frac{n+i}{n} = \frac{52}{2} = 26.0$$

I have been unable to determine the particular relationship between the components of c and the numerical ratio. It seems that they could be related, but that this would need to involve recursion.

[2]There is a problem with the formula as written, however. It only is correct when $c_{\max -1}$ is the ***only*** denomination greater than half of $c_{\max}$. For $c = \{1, 7, 9, 11\}$, our formula predicts an approximation ratio of $\frac{4}{2}$, as above. But BetterChange$(18, c, d)$ returns two coins, an 11¢ and 7¢. Our formula needs to be modified to include all denominations greater than half of $c_{\max}$. Since this won't effect the approximation ratio (i.e. such cases are always less bad than the worst cases) we can ignore this and exclude possible c where there is more than one denomination greater than half of $c_{\max}$.

2

The steps to perform a reversal sort on a sequence π are below.[3]
$\pi = 34658172$
BreakpointReversalSort(π)

1 $\underrightarrow{3\,4}\ \underleftarrow{6\,5}\ \underleftarrow{8}\,\underleftarrow{1}\,\underleftarrow{7}\,\underleftarrow{2}$ b(π)=5
2 $\underrightarrow{3\,4}\ \underleftarrow{6\,5}\ \underleftarrow{8\,7}\ \underrightarrow{1\,2}$ b(π)=3
3 $\underrightarrow{3\,4\,5\,6}\ \underleftarrow{8\,7}\ \underrightarrow{1\,2}$ b(π)=2
4 $\underrightarrow{3\,4\,5\,6\,7\,8}\ \underrightarrow{1\,2}$ b(π)=1
5 $\underrightarrow{3\,4\,5\,6\,7\,8}\ \underleftarrow{2\,1}$ b(π)=1
6 $\underleftarrow{8\,7\,6\,5\,4\,3\,2\,1}$ b(π)=0

It seems there is a shorter method of achieving this sort:
OtherReversalSort(π)

1 $\underrightarrow{3\,4}\ \underleftarrow{6\,5}\ \underleftarrow{8}\,\underleftarrow{1}\,\underleftarrow{7}\,\underleftarrow{2}$ b(π)=5
2 $\underrightarrow{3\,4}\ \underleftarrow{6\,5}\ \underleftarrow{8\,7}\ \underleftarrow{2\,1}$ b(π)=3
3 $\underrightarrow{7\,8}\ \underrightarrow{5\,6}\ \underleftarrow{4\,3\,2\,1}$ b(π)=2
4 $\underrightarrow{7\,8}\ \underleftarrow{6\,5\,4\,3\,2\,1}$ b(π)=1
5 $\underleftarrow{8\,7\,6\,5\,4\,3\,2\,1}$ b(π)=0

This is also a possible outcome of BreakpointReversalSort, and involves making different choices starting at line 3. But the change in b(π) is the same until line 5, where the original sequence I used is unable to decrease b(π).

Is this the shortest possible number of reversals? The smallest theoretical number of reversals r for a permutation p with b number of breakpoints is

$$r_{\min}(p) = \frac{b}{2}$$

This holds because the maximum decrease in breakpoints per reversal is 2. This is a theoretical minimum because it has restrictions on the permutations to which it applies. As mentioned in the book, this can hinge on directionality of the breakpoints. We know that there must always be one breakpoint of each direction for there to be a possibility of decreasing the number of breakpoints (and lack of such is what makes the first reversal sort above not be able to decrease the number of breakpoints from state 4 to state 5). I have not been able to conceive of a non-recursive way of determining the actual number of breakpoints for a given permutation, given the permutation and directionality and length of its breakpoints.

[3] I'm considering a string sorted when it is in either ascending or descending order. Ascending order only would add one reversal each to the reversal sorts below

3

3a. Figure 1 shows s(i, j) plotted for $1 \geq i, j \geq 8$. I plotted by labeling the winning square, s($8, 8$) = L because if a player 'begins' a turn on that square the game is over; the opponent has just won. Because of this, we know that s($8, j$), s($i, 8$) = W because a player beginning her turn there can move to s(8,8). Next, we can infer that s($7, 7$) = L because the only legal moves involve letting the opponent begin their turn on a W square. This scenario is shown in Figure 2. At this point we can notice a pattern: when $i == j$, s(i, j) = L. Otherwise, s(i, j) = W because there exists some legal move from the current space to a space where $i == j$, leaving the opponent only moves where this strategy can be repeated until the endgame. Figure 3 demonstrates this for s($7, 7$). Another way of stating this is that, if $i == j$, it will take at least two rook moves for $(i - n) == (j - n)$. Therefore, the player beginning their turn on a square where $i == j$ will be unable to reach s($8, 8$) unless her opponent makes a mistake.

A recurrence relation for s(i, j) and $max(i, j) \rightarrow min(i, j)$ is

$$s(i,j) = \begin{cases} L \text{ if } i == 8 \wedge j == 8 \\ W \text{ if } \exists s(j \rightarrow 8, i) = L \vee \exists s(j, i \rightarrow 8) = L \\ L \text{ if } \nexists s(j \rightarrow 8, i) = L \wedge \nexists s(j, i \rightarrow 8) = L \end{cases}$$

In this relation, s(8,8) is set to L. All squares that allow a move onto an s(i,j)==L (i.e. $s(7, 8)$ and $s(8, 7)$) are then set to W. All squares that have no L squares in their allowable field of movement are called L. This should proceed recursively from s(8,8) to s(1,1) in a diagonal fashion.

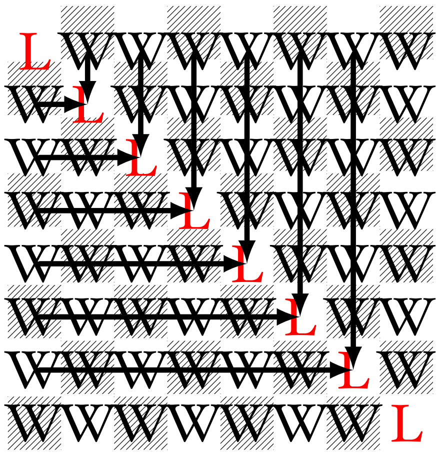

FIGURE 1. s(i, j) for a board of $1 \geq i, j \geq 8$. The squares without arrows indicate a player starting from that square can win immediately by moving to s(i, j). The squares with arrows indicate where the player must move to force a win.

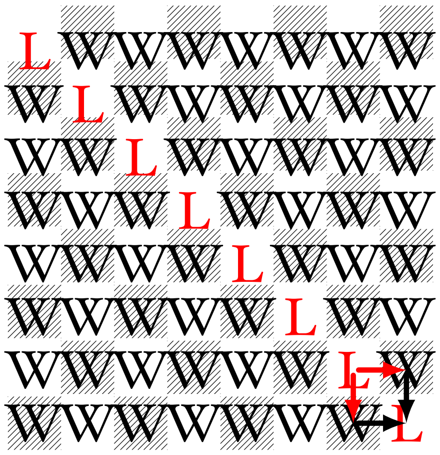

FIGURE 2. An example end game. s(7,7) = L because the player starting from this square's only two legal moves set up the opponent to win, from either s($7, 8$) or s($8, 7$)

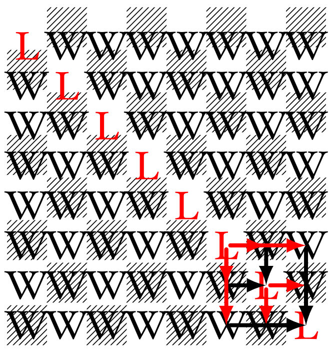

FIGURE 3. s(6, 6) = L because the all moves either allow the opponent to win (i.e. are s($8, j$) or s(i, 8)) or to move the piece to another 'L' square. An annotation of possible moves on the full board would follow the same pattern.

3b. Pseudocode for filling out s(i,j) recursively is shown below.

RookGame(n,m)

```
s = $n \times m$
for i in n to 1
    for j in m to 1
        jwin ← false
        if $i == n \&\& j == m$
            s(i,j) ← L
        else
            for x from j to m
                if s(i,j+x)==L
                    s(i,j) ← W
                    jwin ← true
            if jwin==false
                for x from i to n
                    if s(i+x,j)==L
                        s(i,j) ← W
```

In Figure 1 it's easy to see a pattern: $\begin{cases} L \text{ if } i == j \\ W \end{cases}$

This is the key to making a much faster algorithm.

FastRook(i, j)

```
s = $n \times m$
for i in n to 1
    for j in m to 1
        if i == j
            s(i, j) ← L
        else
            s(i, j) ← W
```

3c & 3d. The first player will always lose if the second player adopts the strategy of moving to a s(i, j) == L, and this is the case iff $i == j$. Because there is one and only one 'L' in each row-column pair, and rooks can travel anywhere along a row or column, it will always be possible to force a win if starting a move on a s(i, j) = W, i.e. for $i \neq j$. So the winning strategy is to go second in this game and always move to a s(i, j) where $i == j$. Eventually, the opponent will be forced to move to either s($8, j$) or s($i, 8$) from which you can move to s($8, 8$) and win, as shown in Figure 2.

4

The following questions will use the sequences shown in Table 1, with a score of -1.

v = TACGGGTAT
w = GGACGTACG

Table 1. Sequences v and w for question 4.

4a. - Global Alignment. A global alignment with backtracking arrows is in Table 2. The alignment has a score of -1, corresponding to a sequence also shown in Table 2.

	T	A	C	G	G	G	T	A	T
G	-1	-2	-3	-4	-5	-6	-7	-8	-9
G	-2↑	-2	-3	-2	-3	-4	-5	-6	-7
A	-3	-1↖	-2	-3	-3	-4	-5	-4	-5
C	-4	-2	0↖	-1	-2	-3	-4	-5	-5
G	-5	-3	-1	1↖	0←	-1 ←	-2	-3	-4
T	-6	-4	-2	0	0	-1	0↖	-1	-2
A	-7	-5	-3	-1	-1	-1	-1	1↖	0
C	-8	-6	-4	-2	-2	-2	-2	0 ↑	0
G	-9	-7	-5	-1	-1	-1	-2	-1	-1↖

v = - T A C G G G T A - T
X X | | | X X | | X X
w = G G A C G- - - T A C G

Table 2. A global alignment of v and w, with a score of -1

4b. - Local Alignment. The optimal local alignment and dynamic programming table are shown in Table 3. This optimal alignment has a score of 4.
4c. Affine Gap Penalty. The affine gap penalty of -20 would bias the alignment away from indels. Also, since it's more than the best possible score of a perfect alignment (10) nor generate anything greater than the worst possible alignment score, -9, in Table 2, this configuration of penalties and bonuses means that the global alignment score is simply the end-to-end alignment of each strand. as shown in Table 4.

An Aside: The question was about global alignments, but it's also interesting to consider local alignments. There are no indels in the local alignment shown in Table 3, so this change in weighting would not have an effect. As an observation, though, weight bonuses of +1 and affine gap penalties of -20 are structured so as to preclude *any* affine gaps in sequences as small as those we have here. This configuration would essentially turn the algorithm into one that searches for the longest consecutive string of matching characters.

	T	A	C	G	G	G	T	A	T
G	0	0	0	1	1	1	0	0	0
G	0	0	0	1	2	2	1	0	0
A	0	1	0	0	1	1	1	2	1
C	0	0	2	1	0	0	0	1	1
G	0	0	1	3	2	1	2	1	1
T	1↘	0	0	2	2	1	2	1	1
A	0	2↘	1	1	1	1	1	3	2
C	0	1	3↘	2	1	0	0	2	2
G	1	0	2	4↘	3	2	1	1	1

```
v =                     T  A  C  G  G  G  T  A  T
                        |  |  |  |
w =  G  G  A  C  G  T  A  C  G
```

Table 3. A local alignment of v and w, with score 4.

```
v =  T  A  C  G  G  G  T  A  T
     X  X  X  X  |  X  X  X  X
w =  G  G  A  C  G  T  A  C  G
```

Table 4. A global alignment of v and w, with score -7.

So, the optimal local alignment remains that in Table 3, with a score of 4.

5

5a & 5b. I cannot think of how to devise an algorithm to determine the edit distance between two strings v and w with time complexity of $O(sn)$ because, in the case where the edit distance is less than s, it will have to traverse a space of $n \times n$ cells, yielding a complexity of $O(n^2)$. Given this, it seems that we can consider 5a and 5b together.

```
LessEdit(v,w,s)
Prepend 0 to v, w
Create Array edit, $|v| \times |w|$
for x ← 1 to |v|
    edit (x,o) ← 0
for x ← 1 to |w|
    edit (o,x) ← 0
for j ← 1 to |v|
    for i ← 1 to |w|
        edit(j,i) ← $\min \begin{cases} \text{edit}(j-1, i-1) - 1 \\ \text{edit}(j, i-1) - 1 \\ \text{edit}(j-1, i) - 1 \\ \text{edit}(j-1, i-1) + 1 \text{ iff } v_{j-1} == w_{i-1} \end{cases}$
        if edit(j,i) $\geq$ s
            return false
return true
```

As discussed above, this algorithm has a time complexity of $O(n^2)$ or $O(nm)$, respectively. This is because of the nested **for** loops. In my research into this topic it seems that Ukkonen developed an edit distance algorithm in 1983 that had a time complexity of $O(nd)$, where d is the edit distance. [4] This algorithm could almost certainly be modified to break when the parameter d s, yielding our solution.

[4]http://www.csse.monash.edu.au/ lloyd/tildeAlgDS/Dynamic/Edit/