COMP 555 PROBLEM SET 1

ELLIOTT HAUSER

1. The chance the rock is black is either 100% or 0%, depending on the starting number of the rocks (which we'll call n_{rocks}). Specifically, if the total number of rocks is even, the final rock will never be black, while if it is odd it will always be black. Let me explain.

Given the rules $\begin{cases} \circ\circ \to \circ \\ \circ\bullet \to \bullet \\ \bullet\bullet \to \circ \end{cases}$ and the requirement that $\frac{n_\circ}{n_\bullet} = \frac{2}{1}$, we can make a few observations from the start. First, as noted in the problem, n_{rocks} will be decremented by one each round of the game. The rules specify the three ways in which this can occur, which we can interpret as increments and/or decrements to $n_\circ$ and $n_\bullet$:

$$\begin{cases} \circ\circ \to \circ \equiv \text{let } n_\circ = n_\circ - 1 \\ \circ\bullet \to \bullet \equiv \text{let } n_\circ = n_\circ - 1 \\ \bullet\bullet \to \circ \equiv \text{let } n_\circ = n_\circ + 1 \text{ and let } n_\bullet = n_\bullet - 2 \end{cases}$$

We can note here that the white rocks can be either incremented or decremented by one. The black rocks are either unchanged or decremented by two. This is the key to solving the problem. The total number of rocks, n_{rocks} is a multiple of $n_\circ + n_\bullet = 3$. In the case where $n_{rocks} = 3$, there will be only one black rock. This rock cannot be removed from the pile, since the rule $\bullet\bullet \to \circ$ will never be invoked, so the last rock in the pile will be black (i.e. the last rule to be invoked will be $\circ\bullet \to \bullet$). In the case where $n_{rocks} = 6$, $n_\circ = 4$ and $n_\bullet = 2$. The black rocks will not be removed until the $\bullet\bullet \to \circ$ rule is invoked, at which point there will be only white rocks in the pile, leaving the final rock white, regardless of the order of rule invocation.

So, to summarize, if n_{rocks} is odd the final rock will be black, whereas if n_{rocks} is even the final rock will be white. If we don't know n_{rocks}, this will yield a 50% chance of the rock being black at the end, mirroring the probability of n_{rocks} being odd, assuming the distribution of n_{rocks} is random amongst its possible values.

Date: September 25, 2012.

2. We can restate Figure 3.4 in the book as a table:

Table 1: Figure 3.4 as a table

Cycle #:	c_0	c_1	c_2	c_3	c_5	c_6
$\frac{whole}{whole}$	1	0	0	0	0	0
$\frac{whole}{x\rightarrow}$	0	1	1	1	1	1
$\frac{x-y}{x\rightarrow}$	0	0	1	2	3	4
$\frac{x-y}{x-y}$	0	0	0	2	8	22
$\frac{\leftarrow y}{x-y}$	0	0	1	2	3	4
$\frac{\leftarrow y}{whole}$	0	1	1	1	1	1
All Molecules	1	2	4	8	16	32

In the table above, $\frac{whole}{whole}$ represents the original DNA molecule, with the two complementary strands on the top and bottom of the line, respectively. Let x and y represent the primer binding sites (consistent with Figure 3.3). So, after the first cycle (c_1), the $\frac{whole}{whole}$ strand is turned into one each of a $\frac{whole}{x\rightarrow}$ and a $\frac{\leftarrow y}{whole}$ molecule. The x and ys here represent the primer bonding sites and the arrows represent the direction of polymerase action. Another cycle (c_2) creates two additional molecules, represented as $\frac{x-y}{x\rightarrow}$ and $\frac{\leftarrow y}{x-y}$. The $x-y$ here indicates that the DNA strand runs from primer bonding site x to y (or y to x, depending on the orientation of the $3'$ and $5'$ ends). Finally, cycle c_3 sees the creation of a $\frac{x-y}{x-y}$ target molecule. This molecule is inert (i.e. once produced is unaffected by subsequent cycles) and so will require a discrete sum as we shall see below.

The number of distinct DNA molecule types, m present at cycle c_n are described by the following:

For any c_n, $\begin{cases} m_{c_n=0} = 1 \\ m_{c_n=1} = 2 \\ m_{c_n=2} = 4 \\ m_{c_n\geq 3} = 5 \end{cases}$ This can be seen by counting the number of nonzero rows

in Table 1. The number of molecules of each type is more complex:

Table 2: Formulas for DNA molecule counts

DNA molecule	count formula*	count at n = 10
$\frac{whole}{whole}$	$n * 0$	0
$\frac{whole}{x\rightarrow}$	$n * 0 + 1$	1
$\frac{x-y}{x\rightarrow}$	$n - 1$	9
$\frac{x-y}{x-y}$	$2^{n-2} + 2(n-2)$	272
$\frac{\leftarrow y}{x-y}$	$n - 1$	9
$\frac{\leftarrow y}{whole}$	$n * 0 + 1$	1
All	$2^{n-2} + 4n - 4$	292

* for $c_{n=2}$ and greater. Table 1 gives the initial counts

The table indicates that, as desired during amplification, the $\frac{x-y}{x-y}$ molecule grows exponentially faster than the others. Thus, though PCR produces DNA molecules other than those desired by researchers, over many cycles the relative frequency of partially amplified sections such as $\frac{\leftarrow y}{x-y}$ will be extremely low.

3. For a long string T and a pattern s of length m, the following pseudocode will find the index within T of the first occurrence of a string s' such that the Hamming distance between s and s' is ≤ 1:

```
WildHamming(T,s)
    i ← 0
    n ← 1
    m ← length(s)
    t ← length(T)
    for j in range(T − m)
        seq ← s
        while i < (m − 1) AND n >= 0
            if T_j != seq_i    # if chars don't match,
                if n −1 < 0          # and we're out of wilds,
                 continue      # next starting point.
                seq_i ← T_j  # otherwise, sub in the char
                    n− −          # and burn a wildcard.
            else
                i + +             # if they match, next item in seq.
             # this loop goes until it reaches the end of seq or a continue.
             # if we make it to the end of seq and don't have negative
             # wild cards, we have a match; return seq.
        if i = (m − 1) AND n >= 0
            return seq
    return false    # if we exit the loop, there's no match.
```

This algorithm is called WildHamming() because it treats the Hamming Distance between the strings as a wildcard, 'using' the wildcard when it encounters a difference between T and s by decrementing n. In addition, when such differences are found, it replaces the character in seq with that of T, which means that seq will be an actual sequence with a Hamming Distance less than n if the program returns. Note that n is hard-coded to 1 for this problem but the algorithm could easily accept a parameter to alter the Hamming distance it uses.

The algorithm will return the seq it has found if it reaches the $m-1$th position in the subset of T it is searching

4a. An $O(n)$ algorithm for compatible1():

```
compatible1(a,ab)
    n=0
    k=0
    for a_i in a
        while k < a_i
            k ← ab_n
            n ++
        if k != a
            return false
        if n = length(ab_n)
            return true
```

This algorithm adds the current and subsequent values of ab until their sum is no longer less than a_i. Then, if ab has been exhausted with no mismatches (which return false), the algorithm returns true. In the case of a mismatch, the fragments of ab and a don't align, meaning they can't have been processed by the same restriction enzyme in their current permutation. The larger algorithm would then attempt another combination of permutations.

It is O(n) because it loops through each item in a and ab only once, without nested loops.

4b. Here is an O(n) complexity algorithm to compute compatible2()

```
compatible2(a, b, AB)

i = 0
j = 0
n = 0

# We can return false right away if the sequences don't add up
if sum(a) != sum(b) != sum(AB)
   return false

while n < sum(AB)
   # Compute ideal ab by calculating increments between a and b
   if (a_{0→i} − n) >= (b_{0→j} − n)
      thisIncrement ← b_j
      if (a_{0→i} − n)== (b_{0→j} − n)
         i++
         j++
      else
         j++
   else
      thisIncrement ← a_i
      i++
   if thisIncrement notIn AB     # Make sure each increment is in AB
      return false          # Ideally, AB should be hashed
   n += thisIncrement          # n keeps track of where we are
return true
```

The algorithm is O(n) because it iterates over a and b only once, and the lookups to a hashed or indexed AB would be constant time.

4c. The branch and bound strategy I'd adopt in the algorithm involves defining the search space and bounding it based on its characteristics with respect to sequence AB.

First the branching. I'd define the search space as every combination of a_i, b_j, and ab_k from i, j, and k to the length of their respective sets, A, B, and AB. Each branch would represent an increment of +1 to k and i, j, or both at once.

Next the bounding. We'd need an incrementer, n, that would keep track of the number of base pairs in ab since the last match was found. This is set to the sum of all double digest pieces from 0 to k in the current ab permutation:

$$n = \sum_{0}^{k} ab_k$$

Then, we must define the desired relationship between a, b, and ab. For every k there must exist some i and j where the difference between n and the running total of a until a_i OR b until b_j is ab_k. Formally,

$$\forall(k)\exists(i,j) \text{ for } ab_k = \begin{cases} \left(\sum_0^i a_i\right) - n \\ \left(\sum_0^j b_j\right) - n \end{cases} \quad \text{where } \triangle k = 1 \text{ \& } \begin{cases} \triangle i = 1, \triangle j = 1 \\ \triangle i = 1, \triangle j = 0 \\ \triangle i = 0, \triangle j = 1 \end{cases}$$

i, j, and k are each integers, starting at zero. This system ensures that each 'chunk' of ab lines up with a corresponding 'chunk' of a or b, even if, for instance, several short chunks of a lined up with ab in a row. If the system is not satisfied, all child permutations of a and b that contain the sequences up to the offending a_i and b_j can be eliminated from the search.

This strategy uses AB as a measuring stick of sorts. Unfortunately, I don't see a way to bound the searches within it. There may be outlier cases where extremely long or short segments in AB will be preclude compatibility with either A or B, which would allow the function to return false much sooner in these cases. It would be an empirical question whether this optimization was worth it, though.

4d. Extremely short segments may be harder to match up, because there will be more compatible permutations of A, B, and AB. This is especially true if both A and B are digested in short segments. On the other hand, if both sets of segments are long, they may be easier to match up, but there will be less information about the sequence. However, if one set of longer segments can provide the easier ordering, while another set of shorter segments can provide more granular information about the sequence, this may be the happy medium. But if the segments produced by A and B are too drastically different in length, it may be difficult to find the correct permutation of the shorter segments, obviating these benefits.

It seems, then, that the segment lengths for A and B should ideally differ, with one being longer than the other, on average, but that the difference in average length should not be so great as to introduce uncertainty into the ordering of the shorter segments. This risk might also be mitigated by keeping the minimum segment length above a certain empirically established threshold.