For the DES challenge, the two most challenging objectives were key generation and determining when/how to check if a key is (potentially) the correct one.

Keyspace and generation

With the key restrictions we are given, we can limit the potential keyspace to a meager 2^35, just shy of 35 billion keys. Many groups chose to split up their key finding by running multiple single-thread processes with different starting points in the keyspace, or similar methods using multi-threaded processes.

```
>>> import itertools
>>> print ["".join(seq) for seq in itertools.product("01",repeat=7)]
['0000000', '0000001', '0000010', ..., '1111101', '1111110', '1111111']
```

Using Python's itertools library, we can find the cartesian product of multiple strings (or, in this example, a single string with the optional 'repeat'). This is equivalent to building a string with nested for loops in a generator expression.

ex:

```
((x,y) for x in A for y in B)
```

or

```
for x in A:
      for y in B:
            (x,y)
```

So now that we have all of the binary permutations of length 7 between 0 (0000000) and 127 (1111111), we can use nested for loops to build the strings in our keyspace. Keep in mind the restriction of $K_i = K_{i+32}$ for i=1-7, 9-15, and 17-23.

```
      perms = ["".join(seq) for seq in itertools.product("01",repeat=7)]
      p = '0'
      for a in perms:
          for b in perms:
              for c in perms:
                  for d in perms:
                      for e in perms:
                          key  = a+p+b+p+c+p+d+p
                          key += a+p+b+p+c+p+e+p
                          print key
```

We can do something similar in Java, let's assume we already have the `perms` array pre-built into a String array, and we will utilize Java's foreach loop to make it easier on us to navigate the multiple nested for loops.

```
for (String a : perms) {
    for (String b : perms) {
        for (String c : perms) {
            for (String d : perms) {
                for (String e : perms) {
                    String key = a+p+b+p+c+p+d+p;
                    key += a+p+b+p+c+p+e+p;
                    System.out.println(key);
                }
            }
        }
    }
}
```

See `deskeys.java` and `deskeys.py` for these implementations.
Note: Don't actually run these! They'll just sit there and print out 35 billion+ keys. But this should give you a good starting point to work on your own key generation algorithm.

Input padding and Key validation
In the assignment description, we are told that each line of 10 Radix-64 characters (60 bits) was padded with four zeros, to make 64 bits. These 60+4 bits are the 64-bit input to DES. Given this fact, we now have a very simple way to tell if a potential key is valid or not by looking at the last 4 bits of the plaintext output. If the last 4 bits do not match '0000' then we know that the key is invalid and we can throw it away. Of course, any random key could potentially give us '0000' for the last 4 bits of the output, so we can use a little bit of probability to increase our chances. By checking multiple blocks of our ciphertext, we can remove (some of) the random probability of a false-positive key. Additionally, any time we decrypt a block that doesn't have the 4 padded zeros, we can immediately throw it away and continue to the next key.

Probability that a random key has 4 0's at the end of decrypted text:

Blocks	Probability
1	1/(2^4) = 1/16
2	1/(2^8) = 1/256
3	1/(2^12) = 4096
4	1/(2^16) = 65536
5	1/(2^20) = 1048576

...	...
10	1/(2^40) = 1099511627776

Using these methods, we can (reasonably) quickly determine the correct key to break the DES cipher.