Core:

Code	Cost	# of times
int j=0	1	1
Character current=pattern.get(j)	1	1
int numMatches=0	1	1
for (int i=0; i<data.size();i++){	1	n*m
if (data.get(i).equals(current)){	1	n
if (j<pattern.size()-1){	1	between 0 and n
j++	1	between 0 and n
current=pattern.get(j) }	1	between 0 and n
else { // pattern complete	0	
UI.println("found at: "+(i-pattern.size()+1))	1	between 0 and n
j=0	1	between 0 and n
current=pattern.get(j)	1	between 0 and n
numMatches++ }}	1	between 0 and n
else {	0	between 0 and n
i=i-j	1	between 0 and n
j=0	1	between 0 and n
current=pattern.get(j) }}	1	between 0 and n
UI.println(numMatches+" exact matches")	1	1

This algorithm is cost O(n*m), this is because if it does not find a correct match, if goes back to index i-j, this means that in the worst case, it will go back m-1 spaces, making the loop n*m.

Completion:
Addition from core code:

```
else if (mismatchesUsed<maxErrors){
    mismatchesUsed++;
    if (j<pattern.size()-1){ //more letters in pattern to search
        j++;
        current=pattern.get(j);
    }
    else { // pattern complete
        String curr="";
        for (int k=i-pattern.size()+1; k<=i; k++){
            curr+=data.get(k);
        }
        UI.printf("found at %d : %s \n",(i-pattern.size()+1),curr);
        j=0;
        current=pattern.get(j);
        numMatches++;
    }
}
```

I believe this algorithm is slightly more expensive, not because of the extra else if statement, as this is done at most 5 times, as that is the most errors it can have, but because of the for loop that is required to print the mismatched string. This can be done at most n/m times. This would make the cost of this algorithm O(n*m+n/m).

Challenge:
This would speed up the search significantly as the first five characters could be found using the Contains key method (O(1)), after that, it would have to only search the indexes that this method returns to see if the rest of the sequence is a match. The cost would therefore go down from O(n*m) to O(m), as searching through the rest of the sequence only costs as there are bases in the sequence.

The speed of the search would be much faster, and more cost effective, however making the map would be the most expensive part. But this is just done once at the start, so it would be worth it if the program is used a lot once it is loaded, as it would become more cost effective with every use when compared to the previous approach.

A map is said to cost 32 * SIZE + 4 * CAPACITY bytes of storage. The size of the map is 1024/ load factor (usually 0.75) = 1366, the capacity is 1024. Therefore the Map would take up 47808 bytes of storage.

The map would take more memory if it was indexed on all 8 bases as there would be more than 1024 options, thus the size would need to be larger to accomodate for the extra entries, adding to the size of the memory needed.

This algorithm would not be particularly useful for the approximate search as it would have to search through the exact search, and then all the possible approximate matches, which could be numerous for multiple errors, and then it would have to account for errors being allowed in the rest of the sequence. Overall, I believe that it would be inefficient, more complex, and potentially more expensive than the current algorithm.

Reference:
http://java-performance.info/memory-consumption-of-java-data-types-2/