The Assignment

We have to implement an advanced string search algorithm capable of finding a given number of consequtive single-byte patterns in an arbitrary string, where the inter-matches distances adhere to given minimums and maximums.

The Specification & Design

I haven't done any further design than what the method in the assignment states, since that's a fully functional and clean method. However, implementing it in a single process would be very slow for DNA research, such as the assignment exemplifies. Research is mostly done on clusters, or at least machines with multiple CPUs, so it would be highly beneficial to parallelize the algorithm.

For simply finding whether the string contains the pattern it would be easiest to split into two threads: one for searching forwards for new matches, and one for searching backwards though previous matches. One could also envision an extra thread that jumped ahead to search on from where the next pattern's lower distance bound is. For finding all matches the algorithm can be split into any number of threads to search sub-strings for sub-patterns and then merge the results.

In this assignment none of these methods would be beneficial, as the overhead of spawning extra threads would only add to the runtime for such short strings.

The Implementation & Proof

For my algorithm I used a well-known implementation of a directed graph (JbDirectedGraph) from http://www.nist.gov/dads/ since there was no need to re-invent that wheel. I have not included a printout of that as it is a by-the-book implementation.

Firstly, proving that the runtime is indeed $O(n^2)$: The first n comes from searching forwards through the string T with "`for(int i=0;i<T.length();i++)`". Then for each match it finds along the way, it has to search backwards through previous matches. In the worst case scenario every letter matches, resulting in n operations that inserts the match in the list of previous matches (done in O(1) time) and a backward search of length n-i. Then for each previous match that is a possible part in a complete match it will insert an edge in the graph with the distance (done in O(1) time), and if at

least q consecutive parts of the pattern has been found it has to search back in the graph at most q steps.

A worst case example: **(¤1)**

T=abcdefghijklmnop (|T| = n = 16)

P={a..p}

L={1..1}

U={1..1}

q=16

This would result in a runtime of 1+2+...+15+16 = 136 plus the final 16 step graph search totalling at 152, with a space usage of 3*16 = 48 for the linked list (containing: value, offset in T, link to previous) of previous matches plus 16 vertices and 15 edges for the graph totalling at 77.

In general terms, runtime is $O(q + \sum_{i=0}^{n} i)$ and space usage is 5n-1. Runtime does not come near $O(n^2)$, which can be explained by the fact that we only have to locate *any* complete match, not *all* complete matches.

The space usage can be further optimized to 4n by killing off the graph and storing the number of longest trail of consecutive preceding matches directly alongside the matches. This would also remove the need to verify the backtrail, thus cutting the q off the runtime. In my implementation I use an array of length n instead of a linked list, which in the worst case is better but generally it is much worse.

Secondly, proving it is actually doing the right thing: I really don't know how I can prove this other than letting the actual code and the above explanation of what it is doing speak for itself. It is much simpler to implement since we only have to find *any* match and the patterns are just single bytes.

The Testing Phase

The assignment states we should design tests before writing the program, which I found quite difficult. The immediate special cases I could think of were q=0, q=1, and q=2. The latter being special since there's no need to build up the graph when all we want is a pair. Another test case would the a worst case example such as the one from above (¤**1**). The program was implemented to handle all these, which it also does. I did not find any extra test cases during or after implementing.

Possible error cases would be if q > n or q > k, or if there are too many or too few lower/upper bounds. However, I have not accounted for malformed input in the program, which results in very low tolerance to such data. It will only tolerate too many lower/upper bounds, simply since it will never try to read past q elements.

Sample Run
algoritmer og datastrukturer er sjovest aieo 4,5,6 11,15,7 3
Output: 1, as expected.

The Conclusion

That we only had to find *any* match from the single-byte patterns simplified the algorithm enormously, which also resulted in a much faster execution time. I was at a loss for proving the algorithm, but its simplicity should make it prove itself. Overall, a peculiar assignment.

Tino Didriksen

The Source

```
import java.io.*;
import java.util.*;

public class StringSearch {
  public static void main(String args[])
  throws FileNotFoundException,IOException {
    FileReader file;
    if (args.length == 0) {
      file = new FileReader("input.txt");
    } else {
      file = new FileReader(args[0]);
    }
    BufferedReader infile = new BufferedReader(file);

    String T  = infile.readLine();
    String P  = infile.readLine();
    String La[] = (infile.readLine()).split(",");
    String Ua[] = (infile.readLine()).split(",");
    int Q    = Integer.parseInt(infile.readLine());

    infile.close();

    int L[] = new int[La.length];
    for(int i=0;i<La.length;i++) {
      L[i] = Integer.parseInt(La[i]);
    }
    La = null;

    int U[] = new int[Ua.length];
    for(int i=0;i<Ua.length;i++) {
      U[i] = Integer.parseInt(Ua[i]);
    }
    Ua = null;

    // We now have T,P,L,U,Q
    System.out.println(patternExists(T,P,L,U,Q) ? "1" : "0");
  }

... continued on next page ...
```

The Source, continued...

```
  public static boolean patternExists
  (String T, String P, int L[], int U[], int Q) {
    final int K = P.length();

    // K*2 vertices seem like a good initial value
    JbDirectedGraph matchGraph = new JbDirectedGraph(K*2);
    int matchList[] = new int[T.length()];

    // Search through T for matches from P
    for(int i=0;i<T.length();i++) {
      matchList[i] = -1; // assume no match
      for(int j=0;j<K;j++) {
        if (T.charAt(i) == P.charAt(j)) {
          // Special case
          if (Q == 1)
            return true;

          matchList[i] = j;

          // Check if there's anything to look back on
          if ((j > 0) && (i > 0)) {
            // Search backwards through previous matches
            for(int k=i-1;k>=0;k--) {
              // Check if it is a preceding part
              if (matchList[k] >= 0 && matchList[k] < j) {
                // Check if it's in valid range
                int d = i-k-1;
                int lower=0,upper=0;
                for(int l=matchList[k];l<j;l++) {
                  lower += L[l];
                  upper += U[l];
                }
                if ((lower <= d) && (d <= upper)) {
                  // Special case
                  if (Q == 2)
                    return true;
                  matchGraph.addVertex(new Integer(k));
                  matchGraph.addVertex(new Integer(i));
                  matchGraph.addEdge(new Integer(i), new Integer(k), d);
                  // Check if there's a chance of a complete match
                  if (j >= Q-1) {
                    Object prev[] = matchGraph.adjacentsOf(new Integer(k));
                    // We know of T[i] and T[k], so h=2 so far.
                    int h=2;
                    // Search for more matches
                    while(prev.length > 0 && prev[0] != null) {
                      prev = matchGraph.adjacentsOf(prev[0]);
                      h++;
                      if (h >= Q)
                        return true;
                    }
                  }
                }
              }
            }
          }
          // Move on to next i
          break;
        }
      }
    }

    return false;
  }
}
```