SACO 2011 FINALS: DAY 1 SOLUTIONS

Finding Squares

(problem and solution by Max Rabkin)

The most important thing to notice here is that the four corners of an axis-aligned square have the coordinates

$$(x, y), (x + s, y), (x, y + s) \text{ and } (x + s, y + s),$$

where (x, y) is the bottom-left corner and s is the side length.

50% solution. This fact allows us to work out if four points form an axis-aligned square. To determine s we can simply take the difference between two coordinates. This allows us to find all the axis-aligned squares with four nested loops over the list of points, by determining whether each 4-tuple of points forms a square. Care must be taken to avoid counting the same square multiple times.

Unfortunately, each nested loop over the list of points multiplies the runtime by N; this can be reduced a little by avoiding looking at the same points multiple times, but this is only enough to get 50%.

100% solution. Wee can speed this up by a factor of almost 4 000 000 in the worst case, by noticing that if we have the coordinates of two corners of a square, we can work out the coordinates of the other two corners. For example, if we have (x, y) and $(x + s, y)$, the other two corners must be at $(x, y + s)$ and $(x + s, y + s)$.

We store the points in a data structure which allows us to quickly determine whether a point is in the set or not (for example, the `set` type in Python and C++, Java's `TreeSet`, or simply a sorted array with a binary search). Then we need only loop over every *pair* of points, and determine whether the other two corners needed to make a square are in the set.

Anagram Escape

(problem and solution by Max Rabkin)

30% solution. The most straightforward solution is to loop through all the length-K substrings of the letter, and determine for each whether it is an anagram of the key. This can be done by sorting the letters in each and determining whether they are equal, or by calculating how many times each character appears in the substring and comparing this to the number of appearances in the key (an array or list of such counts is called a histogram).

This approach takes $N - K + 1$ string comparisons of length involving strings of length K, so in total takes about NK operations. This is fast enough when N is small.

50% solution. We can improve the approach using histograms by using the fact that only two entries in the histogram change when we move from one substring to the next: one character is dropped from the beginning of the substring, so we decrement its count in the histogram, and another is added to the end of the substring, so we increment its count.

To determine whether a substring is an anagram of the key, we compare the entries in the histogram to the pre-calculated histogram of the tree. This comparison takes time proportional to the size of the alphabet, so the running time of the whole algorithm is approximately proportional to N times the size of the alphabet. Therefore, this algorithm will solve 50% of cases in time.

100% solution. The difference between the full solution and the previous one is based on the same observation as before: only two entries of the histogram change at a time when we loop through the substrings of the letter. Therefore, only these two entries can change from being correct (matching the count of the corresponding character in the key) to incorrect or vice versa. Thus, if we keep track of the number of correct entries, we can update it in constant time per loop iteration; likewise, we can check whether the substring is an anagram in constant time per loop iteration: when all the entries are correct, we have found a match.

Language Chains

(problem and solution by Max Rabkin)

The essential problem here is to find the connected components of languages. Once this is done, we can simply train a single translator in one language from each component. One thing to be careful of: if none of the translators speak any languages, it doesn't matter which is chosen, but if some translators know languages and others do not, it is important to choose one who already knows a language (otherwise, one more training than necessary will be required).

Although it is probably simplest to train only a single translator, solutions which use multiple translators are also correct.

50% solution using disjoint sets. There are several solutions which take roughly quadratic time, and score 50%.

One option is to give each language a number representing the component it is in. We start by assigning each language its own component, and then for each translator-language pair, we check if that translator already speaks a language, and if so, merges the component of that language with the component of the current language. To merge component number a and component b, we simply change the component number of every language in component b to a.

This is slow because every time we merge two components, we must loop through all the other languages to possibly change their component number. Even if we keep a list of the languages in a component, so we only loop through the languages in that component, it will still slow down when the component grows large.

50% solution using graphs. Another solution is possibly simpler to code, but requires some basic knowledge of graph theory algorithms. There are many resources on this subject available free online.

We construct a graph of languages, connecting two languages by an edge if there is a translator who speaks both of them. One can then use a graph traversal (e.g., depth-first search) to find the components: if we do not know which component a language is in, we can do a graph traversal starting from that language, marking all languages we encounter as being in the same component.

This algorithm is slow chiefly because of the number of edges it creates: if there is one translator who knows all the languages, it will create $L(L-1)/2$ edges; thus in the larger cases, even creating the graph will take too long.

100% solution using graphs. The previous solution can quite easily be made into a full solution. Instead of connecting every pair of languages that share a translator, one simply connects every language spoken by a translator to a single representative language for that translator. This does not affect the components: it does not matter if two languages are connected via a third language or directly.

This creates a graph with approximately P edges, and a graph traversal will find the number of components in time proportional to $L + P$, which is certainly fast enough.

100% solution using disjoint sets. The idea of disjoint sets can also be used to create a full solution, if one creates a specialised data structure for disjoint sets. For a description of such a structure, see for example `http://en.wikipedia.org/wiki/Union-find`.