Permutations Solution: Modification to Odometer

The permutation problem is as follows: Given a list of items, list all the possible orderings of those items.

Generically, we list permutations as all the orderings of the integers from 0 to n-1, inclusive. For example, if we wanted to list all the permutations for n = 3, in lexicographical ordering, these would be:

0, 1, 2
0, 2, 1
1, 0, 2
1, 2, 0
2, 0, 1
2, 1, 0.

There are several different permutation algorithms, but since recursion an emphasis of the course, a recursive algorithm to solve this problem will be presented. (Feel free to come up with an iterative algorithm on your own.)

We utilize recursion as follows, using the following parameters to our recursive function:

1) An array with a partially filled in permutation.
2) A used array, storing which items have already been partially filled in
3) An integer, k, representing how many items have already been filled in.

Technically, one could have all of this information with just item number 1, it makes life easier to pass in items 2 and 3.
The job of our function will be to list all permutations of the given partially filled in permutation that have their first k values fixed.

Thus, for example, if k = 1 and our partially filled in array looked like:

2		

Then the goal of the algorithm would be to print out (or process in some way) the following two permutations:

2, 0, 1
2, 1, 0

in that order.

As a second example, imagine the following partially filled in array with k = 4 (for permutations with n = 7):

3	6	0	4			

The algorithm should print out the following permutations:

3, 6, 0, 4, 1, 2, 5
3, 6, 0, 4, 1, 5, 2
3, 6, 0, 4, 2, 1, 5
3, 6, 0, 4, 2, 5, 1
3, 6, 0, 4, 5, 1, 2
3, 6, 0, 4, 5, 2, 1

Recursively, our code ought to do the following:

1) Check if k is equal to n, the length of our permutation array. If so, just print out the fully filled in permutation.

2) If not, iterate through each un-used item, placing it in slot k (in numerical order), and recursively calling the function, noting that now, k+1 items are fixed.

In code, we have the following (assume that the appropriate functions and variables are declared):

```
void printperms(vector<int> perm, vector<bool> used, int k) {
    if (k == perm.size()) {
        print(perm);
        return;
    }

    for (int i=0; i<perm.size(); i++) {
        if (!used[i]) {
            used[i] = true;
            perm[k] = i;
            printperms(perm, used, k+1);
            used[i] = false;
        }
    }
}
```

The key here is that because we wanted to quickly know whether or not an item was previously used in the partial solution, adding the used array as a parameter to our method greatly facilitated that decision making process in the code. Of course, the key is that when we use that extra parameter, we MUST keep all information consistent. In this case, that meant properly updating the used array whenever any changes were made to the perm array.

Applying Permutation Algorithm to Objects

Say we wanted to go through all the permutations of some vector of objects, call it items. (Let these be global.) Then we can just do the following:

```
void printperms(vector<int> perm, vector<bool> used, int k) {
    if (k == perm.size()) {
        process(perm);
        return;
    }

    for (int i=0; i<perm.size(); i++) {
        if (!used[i]) {
            used[i] = true;
            perm[k] = i;
            printperms(perm, used, k+1);
            used[i] = false;
        }
    }
}
```

The process function we do whatever it needs to do with the items in the following order:

```
items[perm[0]],
items[perm[1]],
items[perm[2]], …,
items[perm[n-1]]
```

It's fairly simple and common to modify this code so that it returns a value. For example, if we wanted to find the minimum possible answer for a query over all permutations, we would do:

```
int solve(vector<int> perm, vector<bool> used, int k) {
    if (k == perm.size()) return process(perm);
    int res = MAX_VAL;

    for (int i=0; i<perm.size(); i++) {
        if (!used[i]) {
            used[i] = true;
            perm[k] = i;
            int tmp = solve(perm, used, k+1);
            res = min(res, tmp);
            used[i] = false;
        }
    }
    return res;
}
```

Kattis Problem: Ant Typing
Here is a link to the problem:

https://open.kattis.com/problems/anttyping

In this problem we must evaluate the "best" permutation of digits on a keypad. Note that 9! is about 360,000 so we have plenty of time to go through each permutation. But what we can't do is 10^5 work for each permutation.

Notice that the input string is fixed. The ant must type it in order. So all that matters is how long it takes the ant to travel on the keyboard between any pair of digits. Consider this example input:

23734123634

We have each of the following transitions:

23
37
73
34
41
23 (again)
36
63
34 (again)

No matter how long are string is, we'll only have 81 possible transitions (from any one of 9 digits to any of the other 9 digits). Thus, if we precompute how many of each of the 81 transitions there are in the input string, once we have a permutation, we can fairly quickly calculate the total cost of that permutation. Given a permutation, we simply need to calculate the cost of moving between each pair of values.

Here is the solution:

```
using namespace std;
#include <bits/stdc++.h>

// Easier if global.
vector<vector<int>> freq;
string s;

int go(vector<int>& perm, vector<bool>& used, int k);
int eval(const vector<int>& perm);

int main() {

    // Will store how many of each transition is in the input string.
    for (int i=0; i<10; i++)
        freq.push_back(vector<int>(10));
```

```
    // Read input.
    cin >> s;

    // Store all transitions in forward order.
    for (int i=0; i<s.size()-1; i++)
        freq[s[i]-'0'][s[i+1]-'0']++;

    // Set up and solve.
    vector<int> perm(9);
    vector<bool> used(9);
    for (int i=0; i<9; i++) used[i] = false;
    cout << go(perm, used, 0) << endl;
    return 0;
}

// Returns the best answer with the first k items of perm fixed.
int go(vector<int>& perm, vector<bool>& used, int k) {

    if (k == perm.size()) return eval(perm);

    // This is too big, will be overwritten.
    int res = 10000000;

    // Try each possible option in slot k.
    for (int i=0; i<perm.size(); i++) {
        if (used[i]) continue;
        used[i] = true;
        perm[k] = i;
        int tmp = go(perm, used, k+1);
        res = min(res, tmp);
        used[i] = false;
    }

    return res;
}

// Returns the cost of this ordering of the keyboard.
int eval(const vector<int>& perm) {

    vector<vector<int>> costs;
    for (int i=0; i<10; i++) costs.push_back(vector<int>(10));

    // Remember perm is storing 0..8 not 1..9
    for (int i=0; i<perm.size(); i++)
        for (int j=0; j<perm.size(); j++)
            costs[perm[i]+1][perm[j]+1] = abs(i-j);

    int res = 0;
    for (int i=1; i<10; i++)
        for (int j=1; j<10; j++)
            res += (costs[i][j]*freq[i][j]);

    // Add first cost, and typing each one and return.
    res += costs[perm[0]+1][s[0]-'0'];
    return res+s.size();
}
```

There's a lot to unpack here. The beginning part of the code is the precomputation for the freq array, which stores how many of each transition is in the input string. The rest of the permutation code is mostly what would happen in any permutation problem. The eval function is what's unique to this particular problem. We first build the look up table of costs by just doing a double for loop through the permutation and storing each cost explicitly. Once this table is stored, we just have a sum of products to compute over all different types of transitions. Notice that had we looped through the long string (length 10^5), then this code would have taken too much time, almost 1000 times longer, since each transition could appear 1000 times.

Now, let's look at another permutation problem where the items being permuted aren't integers from 0 to n-1. The key for this sort of problem is to keep the permutation algorithm the same, permuting 0 to n-1 only, but then use the permutation array as an index into the array of objects being permuted. These objects themselves should NOT be permuted. This allows us to keep the permutation portion of the code independent and consistent.

Before we get into the last permutation problem, let's discuss a very, very handy C++ function that helps greatly with permutations and avoids the use of recursion:

C++ Function: next_permutation

Given a vector that stores a permutation, the next_permutation function changes the vector to the next permutation. Here is an example of calling the function:

```
next_permutation(vals.begin(),vals.end());
```

The function takes in an iterator to the beginning of the vector and the end of the vector. The function returns a bool. It returns true if the vector was updated, false if the vector was storing the last permutation. Let's use this to solve the last permutation problem in the notes.

Kattis Problem: Class Picture

Here is a link to the problem:

https://open.kattis.com/problems/classpicture

The problem is straightforward. We want to iterate through all possible permutations, and for each one, see if it's possible (see if it has any forbidden pairs next to each other.) If we go through all permutations and never find one that works, then none of them work.

```
using namespace std;
#include <bits/stdc++.h>

// Easier if global.
int n;
vector<string> names;
map<string,int> mymap;
vector<vector<bool>> forbidden;

int main() {

    // Annoying but this is how they do input.
    while (cin >> n) {

        // Read in names and sort.
        names.clear();
        for (int i=0; i<n; i++) {
            string tmp;
            cin >> tmp;
            names.push_back(tmp);
        }
        sort(names.begin(), names.end());

        // Easy lookup.
        mymap.clear();
        for (int i=0; i<n; i++)
            mymap[names[i]] = i;

        // For forbidden positions.
        forbidden.clear();
        for (int i=0; i<n; i++)
            forbidden.push_back(vector<bool>(n));

        // Store all bad pairs.
        int bad;
        cin >> bad;
        for (int i=0; i<bad; i++) {
            string s,t;
            cin >> s >> t;
            forbidden[mymap[s]][mymap[t]] = true;
            forbidden[mymap[t]][mymap[s]] = true;
        }

        // Set up and solve.
        vector<int> perm(n);
        for (int i=0; i<n; i++) perm[i] = i;

        bool hasSol = false;
```

```
        do {

            // See if this permutation is okay.
            bool ok = true;
            for (int i=0; i<n-1; i++) {
                if (forbidden[perm[i]][perm[i+1]]) {
                    ok = false;
                    break;
                }
            }

            // It's okay, indicate that and get out!
            if (ok) {
                hasSol = true;
                break;
            }

        } while (next_permutation(perm.begin(), perm.end()));

        // No sol case.
        if (!hasSol)
            cout << "You all need therapy." << endl;

        // Output names.
        else {
            for (int i=0; i<n-1; i++)
                cout << names[perm[i]] << " ";
            cout << names[perm[n-1]] << endl;
        }

    }

    return 0;
}
```