Brute Force

A computer is much faster than a human, thus, when writing programs to solve problems, we can consider methods of solution that are different than how humans solve problems. As previously mentioned, a C++ program can perform close to 100 million simple instructions in a matter of a second or two (circa 2023).

Thus, while a human might solve a puzzle without trying all of the possible options but instead using logic to reduce the search to a smaller set of "viable options", sometimes it's much easier to write a compute program that just "tries all of the options."

A simple example would be the popular game Wordle. While a human might use logic to figure out what a word might be given some previous clues, a computer program could just loop through all possible words, seeing which were consistent with each given clue.

To write these types of solutions, one must find a way to easily enumerate the whole sample space, the set of prospective solutions. Then, for each prospective solution, write a function to determine if that prospective solution is an actual solution. With Wordle, one must be given a dictionary of valid words to loop through (so this part is easy). Then, one must check each word against each clue (each guess has five clues in the original game, one for each letter guessed), and make sure that the result of each clue matches the prospective word.

We'll take a look at a problem that has a similar type of brute force solution, but one that is much easier to implement.

Kattis Problem: veci

Here is the link to the problem:

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

One way to solve this problem is logically, determining the "next permutation" of the input. This method isn't too difficult actually, and might be the preferred method for many, especially since C++ has a next permutation function.

But, for the purposes of this lecture, to illustrate a simple brute force solution, we'll solve it a different way.

Given an input number, *n*, where *n* has six or fewer digits, we want to find the smallest number greater than *n* with the same exact digits. This means that the answer can be at most one of 10^6 things, easily within the bounds of a brute force search. Thus, the solution is as follows:

For each integer, x, greater than *n*, until we get to 1,000,000, check if x has exactly the same digits as *n*. If so, store the answer and break.

If no such number is found, we answer 0.

It's helpful to write a function that takes in a number and returns a frequency array of its digits, and a second function which compares two frequency arrays to see if they are equal or not.

With these plans in mind, here is a program implementing this solution idea:

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

vector<int> freq(int n);
bool equal(vector<int> a, vector<int> b);

int main() {

    // Read input, get frequencies.
    int n, res = 0;
    cin >> n;
    vector<int> mine = freq(n);

    // Try each possible number.
    for (int i=n+1; i<1000000; i++) {

        // Get this number's frequencies.
        vector<int> other = freq(i);

        // Got it, break!
        if (equal(mine, other)) {
            res = i;
            break;
        }
    }

    cout << res << endl;

    return 0;
}

// Returns a frequency vector of the digits of n.
vector<int> freq(int n) {

    vector<int> res(10);
    for (int i=0; i<10; i++) res[i] = 0;

    // Peel off and count digits.
    while (n > 0) {
        res[n%10]++;
        n /= 10;
    }
    return res;
}

bool equal(vector<int> a, vector<int> b) {
    for (int i=0; i<a.size(); i++)
        if (a[i] != b[i])
            return false;
    return true;
}
```

Another type of brute force pattern is one where we have two options for several choices. For example, consider a problem where we can move Right or Up with each move and we take 5 moves. There are 2^5 possible sequences of moves we could make. Since integers are already stored in binary in the computer, and each bit has two possible values, 0 or 1, we can exploit the bit storage of integers to easily loop through a set of options that represent all possible sequences of choices when we have two choices for each "move." In mathematics, we often think of this sample space as a set of subsets. For example, if we are making 5 moves, each of which is Right or Up, another way of thinking about the sequences of options would be designating each possible subset of moves (label the moves 0, 1, 2, 3 and 4) that are going Right. Here is a list of a few of these subsets, along with the false/true, 0/1, and Up/Right representation of the same set, where we list our items in reverse (4, then 3, then 2, then 1, then 0):

{0, 2, 3}	FTTFT	01101	13	Up, Right, Right, Up, Right
{0, 1}	FFFTT	00011	3	Up, Up, Up, Right, Right
{3}	FTFFF	01000	8	Up, Right, Up, Up, Up
{1, 4}	TFFTF	10010	18	Right, Up, Up, Right, Up

Notice that the 0/1 representation is just binary, and that if we were to loop through all numbers from 0 to 31, we would cycle through all possible settings of five 0s and 1s, which is the same as cycling through all subsets of values set to right.

In order to exploit this fact, we need to be able to easily obtain each separate bit in an integer. We can do this via bitwise operations. Let's take a quick detour to learn several of the bitwise operators and what they do that are provided in C++:

Bitwise Operators in C++

We must be able to easily express a power of two, since the technique above requires us to loop through all non-negative integers upto a power of two minus 1. We can use the left shift operator to do this:

```
1<<n
```

is the number in binary that you obtain when you left shift 1 by n bits. This means placing n 0s to the right of the 1 (moving the 1 to the left by this many bits.) For example,

```
1<<3
```

means `1000` in binary, which is just 2^3 or 8.

More generally, `a<<b` means left-shifting the value of a by b bits. Numerically, unless there is overflow, this is equal to $a2^b$. But, almost always, usually 1 is the number that is left-shifted.

The next bitwise operator is `&`, which is bitwise and. This occurs between two integers. To compute the result, write both out in binary, and in each bit location where both integers have 1s, place a 1, otherwise place a 0.

Here is an example:

```
47 = 101111
29 = 011101
-----------
     001101, so 47&29 = 13
```

Most commonly, the & operator is used to see if a particular bit is on in a number n. Here is the boolean expression which will be true if and only if the k^{th} bit in n is 1:

```
n & (1<<k)
```

or

```
(n & (1<<k)) != 0
```

The former is more succinct, but the latter is probably easier to understand for those used to a programming language other than C/C++.

Basically, the number `(1<<k)` has only one bit set to 1, the k^{th} bit. Thus, when we do a bitwise and, we are guaranteed that all of the other bits in the answer except the k^{th} bit will be set to 0. The k^{th} bit in the answer will be 1 if n's k^{th} is 1 and 0 otherwise. Thus, if this number is 0, the k^{th} bit in n is 0, otherwise it's 1.

The other common bitwise operators are:

```
| - bitwise or
^ - bitwise xor
>> - bitwise right shift
```

A corresponding bit in a bitwise or is 1 if at least one of the input bits is 1.
A corresponding bit in a bitwise xor is 1 if the two bits are different, 0 if they are the same.

So, a bitwise or is good to union two subsets, while a bitwise xor is good at identifying either similarities or differences between two sets, or two binary strings.

Now that we have this background, let's take a look at a problem that can be solved by going through a search space of all subsets.

Kattis Problem: doubleplusgood
Here is the link to the problem:

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

In this problem, for each plus sign, we can either (a) keep it a plus sign, or (b) remove it. Thus, we can think of 0 indicating that we add, and 1 indicating that we fuse the numbers together.

Consider the sample input and the breakdown by each possible subset of signs set to +, with 0 indicating a sign is a plus.

1+9+8+4

000 → 1+9+8+4 = 22
001 → 1+9+84 = 94
010 → 1+98+4 = 103
011 → 1+984 = 985
100 → 19+8+4 = 31
101 → 19+84 = 103
110 → 198+4 = 202
111 → 1984 = 1984

In implementing this solution, we must do the following:

(a) parse the input into separate strings, one for each number.
(b) have a simple way to "fuse" strings of numbers.
(c) use the binary representation of integers to map to when to add/fuse.

Since we might need to "reevaluate" the same sequence of fused strings, let's just precompute a 2D vector of long longs where

```
value[i][j]
```

equals the value of all the strings starting from index i, ending at index j, are equal to. Then we can quickly look up these numeric values when adding, instead of concatenating strings many times.

Pre-computation is useful in brute force when in the process of trying all possibilities, you have to recompute the same item many times.

Putting all of these pieces together, we get the following solution (split code not included):

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

vector<string> split(const string &s, char delim);
```

```
int main() {

    // Read input, split it.
    string line;
    cin >> line;
    vector<string> parts = split(line, '+');
    int n = parts.size();
    vector<vector<ll>> pieces;

    // i is starting string.
    for (int i=0; i<n; i++) {

        // Add to this vector.
        pieces.push_back(vector<ll>(n));

        string tmp = "";

        // j is ending string.
        for (int j=i; j<n; j++) {
            tmp += parts[j];

            // Store this as a ll...so parse it.
            pieces[i][j] = stoll(tmp);
        }
    }

    // Store answers here.
    set<ll> answers;

    // mask represents our subset.
    for (int mask=0; mask<(1<<(n-1)); mask++) {

        // Will store result here.
        ll res = 0;
        int prev = 0;

        // Loop through each bit.
        for (int i=0; i<n; i++) {

            // We are merging numbers.
            if (mask & (1<<i)) continue;

            // Part we add.
            res += pieces[prev][i];

            // Where we start from next.
            prev = i+1;
        }

        // Update our set.
        answers.insert(res);
    }

    // This is all we need.
    cout << answers.size() << endl;
    return 0;
}
```

Now, consider the situation of going through each possible "number" not in base 2, but another base, such as base 3, where there are 3 possible choices/digits at each juncture. We will solve this problem recursively. It's called the "odometer" problem.

The Odometer Problem

The first problem we will encounter is the odometer problem. Given the number of digits in an odometer and the number of possible digits for each slot, print out each possible odometer setting. The problem given above is a specific version of the odometer problem with 4 digits, each with 3 possible choices. To code a solution to the odometer problem, we must write in a recursive function that takes in the array that stores the partial solution, as well as an integer k, representing the number of digits that have already been filled in. The job of the recursive function will be to print out each odometer setting with the first k digits from the array fixed as they are. Using the prior example, odometer([2, 0, …, …], 2) should print out the 9 odometer settings previously shown that all start with 2, 0.

Here is the method:

```
void odometer(vector<int> digits, int numDigits, int k) {

    if (k == digits.size()) {
        print(digits);
        return;
    }

    for (int i=0; i<numDigits; i++) {
        digits[k] = i;
        odometer(digits, k+1);
    }
}
```

In short, what we do is as follows in the recursive case:

For each possible item that could go in slot k, place it there, and recursively print out all solutions with those first k+1 items fixed.

For the odometer problem, it's very simple to figure out the possible items that could go in slot k. They are ALWAYS the items 0 through NUMDIGITS-1.

In our other problems, this task (determining what is allowed to go in slot k), is more difficult and may require more information to be passed into the method. A relatively slight edit to the odometer problem leads us to a solution to the ubiquitous combination and permutation problems. Since we already covered the solution to the combinations problem, we will move onto the permutations problem.

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;
}
```