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.