

The following is a slight modification to the code:

```
char input[100],ch;
void main(void)
{
    freopen("input.txt","rb",stdin);
    scanf("%s\n",&input);
    scanf("%c",&ch);
}
```

What will be their value now? The value of `ch` will be `'\n'` for the first code and `'d'` for the second code.

Be careful about using `gets()` and `scanf()` together !

You should also be careful about using `gets()` and `scanf()` in the same program. Test it with the following scenario. The code is:

```
scanf("%s\n",&dummy);
gets(name);
```

And the input file is:

```
ABCDEF
bbbbbbXXX
```

What do you get as the value of `name`? `"XXX"` or `"bbbbbbXXX"` (Here, `"b"` means blank or space)

Multiple input programs

"Multiple input programs" are an invention of the online judge. The online judge often uses the problems and data that were first presented in live contests. Many solutions to problems presented in live contests take a single set of data, give the output for it, and terminate. This does not imply that the judges will give only a single set of data. The judges actually give multiple files as input one after another and compare the corresponding output files with the judge output. However, the Valladolid online judge gives only one file as input. It inserts all the judge inputs into a single file and at the top of that file, it writes how many sets of inputs there are. This number is the same as the number of input files the contest judges used. A blank line now separates each set of data. So the structure of the input file for multiple input program becomes:

```
Integer N      //denoting the number of sets of input
--blank line---
input set 1 //As described in the problem statement
--blank line---

input set 2 //As described in the problem statement
--blank line---
input set 3 //As described in the problem statement
--blank line---
.
.
.
--blank line---
input set n //As described in the problem statement
--end of file--
```

Note that there should be no blank after the last set of data. The structure of the output file for a multiple input program becomes:

```
Output for set 1 //As described in the problem statement
--Blank line---
Output for set 2 //As described in the problem statement
--Blank line---
Output for set 3 //As described in the problem statement
--Blank line---
.
.
.
--blank line---
Output for set n //As described in the problem statement
--end of file--
```

The USU online judge does not have multiple input programs like Valladolid. It prefers to give multiple files as input and sets a time limit for each set of input.

Problems of multiple input programs

There are some issues that you should consider differently for multiple input programs. Even if the input specification says that the input terminates with the end of file (EOF), each set of input is actually terminated by a blank line, except for the last one, which is terminated by the end of file. Also, be careful about the initialization of variables. If they are not properly initialized, your program may work for a single set of data but give correct output for multiple sets of data. All global variables are initialized to their corresponding zeros.^[6]

CHAPTER 6 BRUTE FORCE METHOD

This is the most basic problem solving technique. Utilizing the fact that computer is actually very fast.

Complete search exploits the brute force, straight-forward, try-them-all method of finding the answer. This method should almost always be the first algorithm/solution you consider. If this works within time and space constraints, then do it: it's easy to code and usually easy to debug. This means you'll have more time to work on all the hard problems, where brute force doesn't work quickly enough.

Party Lamps

You are given N lamps and four switches. The first switch toggles all lamps, the second the even lamps, the third the odd lamps, and last switch toggles lamps 1,4,7,10,...

Given the number of lamps, N , the number of button presses made (up to 10,000), and the state of some of the lamps (e.g., lamp 7 is off), output all the possible states the lamps could be in.

Naively, for each button press, you have to try 4 possibilities, for a total of 4^{10000} (about 10^{6020}), which means there's no way you could do complete search (this particular algorithm would exploit recursion).

Noticing that the order of the button presses does not matter gets this number down to about 10000^4 (about 10^{16}), still too big to completely search (but certainly closer by a factor of over 10^{6000}).

However, pressing a button twice is the same as pressing the button no times, so all you really have to check is pressing each button either 0 or 1 times. That's only $2^4 = 16$ possibilities, surely a number of iterations solvable within the time limit.

The Clocks

A group of nine clocks inhabits a 3×3 grid; each is set to 12:00, 3:00, 6:00, or 9:00. Your goal is to manipulate them all to read 12:00. Unfortunately, the only way you can manipulate the clocks is by one of nine different types of move, each one of which rotates a certain subset of the clocks 90 degrees clockwise. Find the shortest sequence of moves which returns all the clocks to 12:00.

The 'obvious' thing to do is a recursive solution, which checks to see if there is a solution of 1 move, 2 moves, etc. until it finds a solution. This would take 9^k time, where k is the number of moves. Since k might be fairly large, this is not going to run with reasonable time constraints.

Note that the order of the moves does not matter. This reduces the time down to k^9 , which isn't enough of an improvement.

However, since doing each move 4 times is the same as doing it no times, you know that no move will be done more than 3 times. Thus, there are only 49 possibilities, which is only 262,072, which, given the rule of thumb for run-time of more than 10,000,000 operations in a second, should work in time. The brute-force solution, given this insight, is perfectly adequate.

It is beautiful, isn't it? If you have this kind of reasoning ability. Many seemingly hard problems is eventually solvable using brute force.

Recursion

Recursion is cool. Many algorithms need to be implemented recursively. A popular combinatorial brute force algorithm, backtracking, usually implemented recursively. After highlighting it's importance, lets study it.

Recursion is a function/procedure/method that calls itself again with a smaller range of arguments (break a problem into simpler problems) or with different arguments (it is useless if you use recursion with the same arguments). It keeps calling itself until something that is so simple / simpler than before (which we called base case), that can be solved easily or until an exit condition occurs.

A recursion must stop (actually, all program must terminate). In order to do so, a valid base case or end-condition must be reached. Improper coding will leads to stack overflow error.

You can determine whether a function recursive or not by looking at the source code. If in a function or procedure, it calls itself again, it's recursive type.

When a method/function/procedure is called:

- caller is suspended,
- "state" of caller saved,
- new space allocated for variables of new method.

Recursion is a powerful and elegant technique that can be used to solve a problem by solving a smaller problem of the same type. Many problems in Computer Science involve recursion and some of them are naturally recursive.

If one problem can be solved in both way (recursive or iterative), then choosing iterative version is a good idea since it is faster and doesn't consume a lot of memory. Examples: Factorial, Fibonacci, etc.

However, there are also problems that are can only be solved in recursive way or more efficient in recursive type or when it's iterative solutions are difficult to conceptualize. Examples: Tower of Hanoi, Searching (DFS, BFS), etc.

Types of recursion

There are 2 types of recursion, Linear recursion and multiple branch (Tree) recursion.

1. Linear recursion is a recursion whose order of growth is linear (not branched). Example of Linear recursion is Factorial, defined by $fac(n) = n * fac(n-1)$.

2. Tree recursion will branch to more than one node each step, growing up very quickly. It provides us a flexible ways to solve some logically difficult problem. It can be used to perform a Complete Search (To make the computer to do the trial and error). This recursion type has a quadratic or cubic or more order of growth and therefore, it is not suitable for solving "big" problems. It's limited to small. Examples: solving Tower of Hanoi, Searching (DFS, BFS), Fibonacci number, etc.

Some compilers can make some type of recursion to be iterative. One example is tail-recursion elimination. Tail-recursive is a type of recursive which the recursive call is the last command in that function/procedure. This

Tips on Recursion

1. Recursion is very similar to mathematical induction.
2. You first see how you can solve the base case, for $n=0$ or for $n=1$.
3. Then you assume that you know how to solve the problem of size $n-1$, and you look for a way of obtaining the solution for the problem size n from the solution of size $n-1$.

When constructing a recursive solution, keep the following questions in mind:

1. How can you define the problem in term of a smaller problem of the same type?
2. How does each recursive call diminish the size of the problem?

3. What instance of the problem can serve as the base case?
4. As the problem size diminishes, will you reach this base case?

Sample of a very standard recursion, Factorial (in Java):

```
static int factorial(int n) {  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

Divide and Conquer

This technique use analogy "smaller is simpler". Divide and conquer algorithms try to make problems simpler by dividing it to sub problems that can be solved easier than the main problem and then later it combine the results to produce a complete solution for the main problem.

Divide and Conquer algorithms: Quick Sort, Merge Sort, Binary Search.

Divide & Conquer has 3 main steps:

1. Divide data into parts
2. Find sub-solutions for each of the parts recursively (usually)
3. Construct the final answer for the original problem from the sub-solutions

```
DC(P) {  
    if small(P) then  
        return Solve(P);  
    else {  
        divide P into smaller instances P1,...,Pk, k>1;  
        Apply DC to each of these subproblems;  
        return combine(DC(P1),...,DC(Pk));  
    }  
}
```

Binary Search, is not actually follow the pattern of the full version of Divide & Conquer, since it never combine the sub problems solution anyway. However, lets use this example to illustrate the capability of this technique.

Binary Search will help you find an item in an array. The naive version is to scan through the array one by one (sequential search). Stopping when we found the item (success) or when we reach the end of array (failure). This is the simplest and the most inefficient way

to find an element in an array. However, you will usually end up using this method because not every array is ordered, you need sorting algorithm to sort them first.

Binary search algorithm is using Divide and Conquer approach to reduce search space and stop when it found the item or when search space is empty.

Pre-requisite to use binary search is the array must be an ordered array, the most commonly used example of ordered array for binary search is searching an item in a dictionary.

Efficiency of binary search is $O(\log n)$. This algorithm was named "binary" because its behavior is to divide search space into 2 (binary) smaller parts).

Optimizing your source code

Your code must be fast. If it cannot be "fast", then it must at least "fast enough". If time limit is 1 minute and your currently program run in 1 minute 10 seconds, you may want to tweak here and there rather than overhaul the whole code again.

Generating vs Filtering

Programs that generate lots of possible answers and then choose the ones that are correct (imagine an 8-queen solver) are filters. Those that hone in exactly on the correct answer without any false starts are generators. Generally, filters are easier (faster) to code and run slower. Do the math to see if a filter is good enough or if you need to try and create a generator.

Pre-Computation / Pre-Calculation

Sometimes it is helpful to generate tables or other data structures that enable the fastest possible lookup of a result. This is called Pre-Computation (in which one trades space for time). One might either compile Pre-Computed data into a program, calculate it when the program starts, or just remember results as you compute them. A program that must translate letters from upper to lower case when they are in upper case can do a very fast table lookup that requires no conditionals, for example. Contest problems often use prime numbers - many times it is practical to generate a long list of primes for use elsewhere in a program.

Decomposition

While there are fewer than 20 basic algorithms used in contest problems, the challenge of combination problems that require a combination of two algorithms for solution is

daunting. Try to separate the cues from different parts of the problem so that you can combine one algorithm with a loop or with another algorithm to solve different parts of the problem independently. Note that sometimes you can use the same algorithm twice on different (independent!) parts of your data to significantly improve your running time.

Symmetries

Many problems have symmetries (e.g., distance between a pair of points is often the same either way you traverse the points). Symmetries can be 2-way, 4-way, 8-way, and more. Try to exploit symmetries to reduce execution time.

For instance, with 4-way symmetry, you solve only one fourth of the problem and then write down the four solutions that share symmetry with the single answer (look out for self-symmetric solutions which should only be output once or twice, of course).

Solving forward vs backward

Surprisingly, many contest problems work far better when solved backwards than when using a frontal attack. Be on the lookout for processing data in reverse order or building an attack that looks at the data in some order or fashion other than the obvious.

CHAPTER 7 MATHEMATICS

Base Number Conversion

Decimal is our most familiar base number, whereas computers are very familiar with binary numbers (octal and hexa too). The ability to convert between bases is important. Refer to mathematics book for this.^[2]

TEST YOUR BASE CONVERSION KNOWLEDGE

Solve UVa problems related with base conversion:

[343 - What Base Is This?](#)

[353 - The Bases Are Loaded](#)

[389 - Basically Speaking](#)

Big Mod

Modulo (remainder) is important arithmetic operation and almost every programming language provide this basic operation. However, basic modulo operation is not sufficient if we are interested in finding a modulo of a very big integer, which will be difficult and has tendency for overflow. Fortunately, we have a good formula here:

$$(A*B*C) \bmod N == ((A \bmod N) * (B \bmod N) * (C \bmod N)) \bmod N.$$

To convince you that this works, see the following example:

`(7*11*23) mod 5 is 1; let's see the calculations step by step:`

```
((7 mod 5) * (11 mod 5) * (23 mod 5)) Mod 5 =
((2 * 1 * 3) Mod 5) =
(6 Mod 5) = 1
```

This formula is used in several algorithm such as in Fermat Little Test for primality testing:

$R = B^P \bmod M$ (B^P is a **very very big** number). By using the formula, we can get the correct answer without being afraid of overflow.

This is how to calculate **R** quickly (using Divide & Conquer approach, see exponentiation below for more details):

```

long bigmod(long b, long p, long m) {
    if (p == 0)
        return 1;
    else if (p%2 == 0)
        return square(bigmod(b, p/2, m)) % m; // square(x) = x * x
    else
        return ((b % m) * bigmod(b, p-1, m)) % m;
}

```

TEST YOUR BIG MOD KNOWLEDGE

Solve UVa problems related with Big Mod:

[374 - Big Mod](#)

[10229 - Modular Fibonacci](#) - plus Fibonacci

Big Integer

Long time ago, 16-bit integer is sufficient: $[-2^{15} \text{ to } 2^{15-1}] \sim [-32,768 \text{ to } 32,767]$.

When 32-bit machines are getting popular, 32-bit integers become necessary. This data type can hold range from $[-2^{31} \text{ to } 2^{31-1}] \sim [-2,147,483,648 \text{ to } 2,147,483,647]$. Any integer with 9 or less digits can be safely computed using this data type. If you somehow must use the 10th digit, be careful of overflow.

But man is very greedy, 64-bit integers are created, referred as programmers as long long data type or Int64 data type. It has an awesome range up that can covers 18 digits integers fully, plus the 19th digit partially $[-2^{63}-1 \text{ to } 2^{63-1}] \sim [-9,223,372,036,854,775,808 \text{ to } 9,223,372,036,854,775,807]$. Practically this data type is safe to compute most of standard arithmetic problems, except some problems.

Note: for those who don't know, in C, long long n is read using: `scanf("%lld",&n);` and unsigned long long n is read using: `scanf("%llu",&n);` **For 64 bit data:** `typedef unsigned long long int64; int64 test_data=64000000000LL`

Now, RSA cryptography need 256 bits of number or more. This is necessary, human always want more security right? However, some problem setters in programming contests also follow this trend. Simple problem such as finding n'th factorial will become very very hard once the values exceed 64-bit integer data type. Yes, long double can store very high numbers, but it actually only stores first few digits and an exponent, long double is not precise.

Implement your own arbitrary precision arithmetic algorithm. Standard pen and pencil algorithm taught in elementary school will be your tool to implement basic arithmetic operations: addition, subtraction, multiplication and division. More sophisticated

algorithm are available to enhance the speed of multiplication and division. Things are getting very complicated now.

TEST YOUR BIG INTEGER KNOWLEDGE

Solve UVa problems related with Big Integer:

[485 - Pascal Triangle of Death](#)
[495 - Fibonacci Freeze](#) - plus Fibonacci
[10007 - Count the Trees](#) - plus Catalan formula
[10183 - How Many Fibs](#) - plus Fibonacci
[10219 - Find the Ways !](#) - plus Catalan formula
[10220 - I Love Big Numbers !](#) - plus Catalan formula
[10303 - How Many Trees?](#) - plus Catalan formula
[10334 - Ray Through Glasses](#) - plus Fibonacci
[10519 - !!Really Strange!!](#)
[10579 - Fibonacci Numbers](#) - plus Fibonacci

Carmichael Number

Carmichael number is a number which is not prime but has ≥ 3 prime factors. You can compute them using prime number generator and prime factoring algorithm.

The first 15 Carmichael numbers are:

561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657, 52633, 62745, 63973

TEST YOUR CARMICHAEL NUMBER KNOWLEDGE

Solve UVa problems related with Carmichael Number:

[10006 - Carmichael Number](#)

Catalan Formula

The number of distinct binary trees can be generalized as Catalan formula, denoted as $C(n)$.

$$C(n) = \frac{2n}{n+1} C_{n-1}$$

If you are asked values of several $C(n)$, it may be a good choice to compute the values bottom up.

Since $C(n+1)$ can be expressed in $C(n)$, as follows:

$$\begin{aligned}
 \text{Catalan}(n) &= \frac{2n!}{n! * n! * (n+1)} \\
 \text{Catalan}(n+1) &= \frac{2 * (n+1)}{(n+1)! * (n+1)! * ((n+1)+1)} = \\
 &= \frac{(2n+2) * (2n+1) * 2n!}{(n+1) * n! * (n+1) * n! * (n+2)} = \\
 &= \frac{(2n+2) * (2n+1) * 2n!}{(n+1) * (n+2) * n! * n! * (n+1)} = \\
 &= \frac{(2n+2) * (2n+1)}{(n+1) * (n+2)} * \text{Catalan}(n)
 \end{aligned}$$

TEST YOUR CATALAN FORMULA KNOWLEDGE

Solve UVa problems related with Catalan Formula:

[10007 - Count the Trees](#) - * n! -> number of trees + its permutations
[10303 - How Many Trees?](#)

Counting Combinations - $C(N,K)$

$C(N,K)$ means how many ways that N things can be taken K at a time. This can be a great challenge when N and/or K become very large.

Combination of (N,K) is defined as:

$$\frac{N!}{(N-K)! * K!}$$

For your information, the exact value of $100!$ is:

```
93,326,215,443,944,152,681,699,238,856,266,700,490,715,968,264,381,621,46
8,592,963,895,217,599,993,229,915,608,941,463,976,156,518,286,253,697,920
,827,223,758,251,185,210,916,864,000,000,000,000,000,000,000,000
```

So, how to compute the values of $C(N,K)$ when N and/or K is big but the result is guaranteed to fit in 32-bit integer?

Divide by GCD before multiply (sample code in C)

```
long gcd(long a,long b) {
    if (a%b==0) return b; else return gcd(b,a%b);
}
void Divbygcd(long& a,long& b) {
    long g=gcd(a,b);
    a/=g;
    b/=g;
}
long C(int n,int k){
    long numerator=1,denominator=1,toMul,toDiv,i;
    if (k>n/2) k=n-k; /* use smaller k */
    for (i=k;i;i--) {
        toMul=n-k+i;
        toDiv=i;
        Divbygcd(toMul,toDiv);      /* always divide before multiply */
        Divbygcd(numerator,toDiv);
    }
    Divbygcd(toMul,denominator);
    numerator*=toMul;
    denominator*=toDiv;
}
return numerator/denominator;
}
```

TEST YOUR $C(N,K)$ KNOWLEDGE

Solve UVa problems related with Combinations:

[369 - Combinations](#)

[530 - Binomial Showdown](#)

Divisors and Divisibility

If d is a divisor of n , then so is n/d , but d & n/d cannot both be greater than \sqrt{n} .

```
2 is a divisor of 6, so 6/2 = 3 is also a divisor of 6
```

Let $N = 25$, Therefore no divisor will be greater than $\sqrt{25} = 5$.
(Divisors of 25 is 1 & 5 only)

If you keep that rule in your mind, you can design an algorithm to find a divisor better, that's it, no divisor of a number can be greater than the square root of that particular number.

If a number $N = a^i * b^j * \dots * c^k$ then N has $(i+1)*(j+1)*\dots*(k+1)$ divisors.

TEST YOUR DIVISIBILITY KNOWLEDGE

Solve UVa problems related with Divisibility:

[294 - Divisors](#)

Exponentiation

(Assume `pow()` function in `<math.h>` doesn't exist...). Sometimes we want to do exponentiation or take a number to the power n . There are many ways to do that. The standard method is standard multiplication.

Standard multiplication

```
long exponent(long base, long power) {
    long i, result = 1;
    for (i=0; i<power; i++) result *= base;
    return result;
}
```

This is 'slow', especially when power is big - $O(n)$ time. It's better to use divide & conquer.

Divide & Conquer exponentiation

```
long square(long n) { return n*n; }
long fastexp(long base, long power) {
    if (power == 0)
        return 1;
    else if (power%2 == 0)
        return square(fastexp(base, power/2));
    else
        return base * (fastexp(base, power-1));
}
```

Using built in formula, $a^n = \exp(\log(a)*n)$ or $\text{pow}(a,n)$

```
#include <stdio.h>
#include <math.h>

void main() {
    printf("%lf\n",exp(log(8.0)*1/3.0));
    printf("%lf\n",pow(8.0,1/3.0));
}
```

TEST YOUR EXPONENTIATION KNOWLEDGE

Solve UVa problems related with Exponentiation:

[113 - Power of Cryptography](#)

Factorial

Factorial is naturally defined as $\text{Fac}(n) = n * \text{Fac}(n-1)$.

Example:

```
Fac(3) = 3 * Fac(2)
Fac(3) = 3 * 2 * Fac(1)
Fac(3) = 3 * 2 * 1 * Fac(0)
Fac(3) = 3 * 2 * 1 * 1
Fac(3) = 6
```

Iterative version of factorial

```
long FacIter(int n) {
    int i,result = 1;
    for (i=0; i<n; i++) result *= i;
    return result;
}
```

This is the best algorithm to compute factorial. $O(n)$.

Fibonacci

Fibonacci numbers was invented by Leonardo of Pisa (Fibonacci). He defined fibonacci numbers as a growing population of immortal rabbits. Series of Fibonacci numbers are: 1,1,2,3,5,8,13,21,...

Recurrence relation for Fib(n):

```
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1) + Fib(n-2)
```

Iterative version of Fibonacci (using Dynamic Programming)

```
int fib(int n) {
    int a=1,b=1,i,c;
    for (i=3; i<=n; i++) {
        c = a+b;
        a = b;
        b = c;
    }
    return a;
}
```

This algorithm runs in linear time $O(n)$.

Quick method to quickly compute Fibonacci, using Matrix property.

```
Divide_Conquer_Fib(n) {
    i = h = 1;
    j = k = 0;
    while (n > 0) {
        if (n%2 == 1) { // if n is odd
            t = j*h;
            j = i*h + j*k + t;
            i = i*k + t;
        }
        t = h*h;
        h = 2*k*h + t;
        k = k*k + t;
        n = (int) n/2;
    } return j;}
}
```

This runs in $O(\log n)$ time.

TEST YOUR FIBONACCI KNOWLEDGE

Solve UVa problems related with Fibonacci:

[495 - Fibonacci Freeze](#)

[10183 - How Many Fibs](#)

[10229 - Modular Fibonacci](#)

[10334 - Ray Through Glasses](#)

[10450 - World Cup Noise](#)

[10579 - Fibonacci Numbers](#)

Greatest Common Divisor (GCD)

As its name suggests, Greatest Common Divisor (GCD) algorithm finds the greatest common divisor between two numbers a and b . GCD is a very useful technique, for example to reduce rational numbers into its smallest version ($3/6 = 1/2$). The best GCD algorithm so far is Euclid's Algorithm. Euclid found this interesting mathematical equality: $\text{GCD}(a,b) = \text{GCD}(b, (a \bmod b))$. Here is the fastest implementation of GCD algorithm

```
int GCD(int a,int b) {
    while (b > 0) {
        a = a % b;
        a ^= b;    b ^= a;    a ^= b;    }    return a;
}
```

Lowest Common Multiple (LCM)

Lowest Common Multiple and Greatest Common Divisor (GCD) are two important number properties, and they are interrelated. Even though GCD is more often used than LCM, it is useful to learn LCM too.

```
LCM (m,n) = (m * n) / GCD (m,n)

LCM (3,2) = (3 * 2) / GCD (3,2)
LCM (3,2) = 6 / 1
LCM (3,2) = 6
```

Application of LCM -> to find a synchronization time between two traffic lights, if traffic light A displays green color every 3 minutes and traffic light B displays green color every 2 minutes, then every 6 minutes, both traffic lights will display green color at the same time.

Mathematical Expressions

There are three types of mathematical/algebraic expressions. They are Infix, Prefix, & Postfix expressions.

Infix expression grammar:

```
<infix> = <identifier> | <infix><operator><infix>
<operator> = + | - | * | / | <identifier> = a | b | .... | z
```

Infix expression example: $(1 + 2) \Rightarrow 3$, the operator is in the middle of two operands. Infix expression is the normal way humans compute numbers.

Prefix expression grammar:

$\langle \text{prefix} \rangle = \langle \text{identifier} \rangle \mid \langle \text{operator} \rangle \langle \text{prefix} \rangle \langle \text{prefix} \rangle$
 $\langle \text{operator} \rangle = + \mid - \mid * \mid / \mid \langle \text{identifier} \rangle = a \mid b \mid \dots \mid z$

Prefix expression example: $(+ 1 2) \Rightarrow (1 + 2) = 3$, the operator is the first item.

One programming language that use this expression is Scheme language.

The benefit of prefix expression is it allows you write: $(1 + 2 + 3 + 4)$ like this $(+ 1 2 3 4)$, this is simpler.

Postfix expression grammar:

$\langle \text{postfix} \rangle = \langle \text{identifier} \rangle \mid \langle \text{postfix} \rangle \langle \text{postfix} \rangle \langle \text{operator} \rangle$
 $\langle \text{operator} \rangle = + \mid - \mid * \mid / \mid \langle \text{identifier} \rangle = a \mid b \mid \dots \mid z$

Postfix expression example: $(1 2 +) \Rightarrow (1 + 2) = 3$, the operator is the last item.

Postfix Calculator

Computer do postfix calculation better than infix calculation. Therefore when you compile your program, the compiler will convert your infix expressions (most programming language use infix) into postfix, the computer-friendly version.

Why do computer like Postfix better than Infix?

It's because computer use stack data structure, and postfix calculation can be done easily using stack.

Infix to Postfix conversion

To use this very efficient Postfix calculation, we need to convert our Infix expression into Postfix. This is how we do it:

Example:

Infix statement: $((4 - (1 + 2 * (6 / 3) - 5)))$. This is what the algorithm will do, look carefully at the steps.

Expression	Stack (bottom to top)	Postfix expression
$((4 - (1 + 2 * (6 / 3) - 5)))$	(
$((4 - (1 + 2 * (6 / 3) - 5)))$	((
$((4 - (1 + 2 * (6 / 3) - 5)))$	((4
$((4 - (1 + 2 * (6 / 3) - 5)))$	((-	4

((4-(1+2*(6/3)-5)))	(((-(4
((4-(1+2*(6/3)-5)))	(((-(41
((4-(1+2*(6/3)-5)))	(((-(+	41
((4-(1+2*(6/3)-5)))	(((-(+	412
((4-(1+2*(6/3)-5)))	(((-(+	412
((4-(1+2*(6/3)-5)))	(((-(+(412
((4-(1+2*(6/3)-5)))	(((-(+(4126
((4-(1+2*(6/3)-5)))	(((-(+(/	4126
((4-(1+2*(6/3)-5)))	(((-(+(/	41263
((4-(1+2*(6/3)-5)))	(((-(+(41263/
((4-(1+2*(6/3)-5)))	(((-(+(41263/*+
((4-(1+2*(6/3)-5)))	(((-(+(41263/*+5
((4-(1+2*(6/3)-5)))	(((-(+(41263/*+5-
((4-(1+2*(6/3)-5)))	(((-(+(41263/*+5--
((4-(1+2*(6/3)-5)))	(((-(+(41263/*+5--

TEST YOUR INFIX->POSTFIX KNOWLEDGE

Solve UVa problems related with postfix conversion:

727 - Equation

Prime Factors

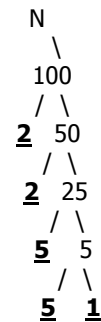
All integers can be expressed as a product of primes and these primes are called **prime factors** of that number.

Exceptions:

For negative integers, multiply by -1 to make it positive again.
For -1,0, and 1, no prime factor. (by definition...)

Standard way

Generate a prime list again, and then check how many of those primes can divide n. This is very slow. Maybe you can write a very efficient code using this algorithm, but there is another algorithm that is much more effective than this.



Creative way, using Number Theory

1. Don't generate prime, or even checking for primality.
2. Always use **stop-at-sqrt** technique
3. Remember to check repetitive prime factors, example= $20 \rightarrow 2 \cdot 2 \cdot 5$, don't count 2 twice. We want distinct primes (however, several problems actually require you to count these multiples)
4. Make your program use constant memory space, no array needed.
5. Use the definition of prime factors wisely, this is the key idea. A number can always be divided into a prime factor and another prime factor or another number. This is called the factorization tree.

From this factorization tree, we can determine these following properties:

1. If we take out a prime factor (F) from any number N, then the N will become smaller and smaller until it become 1, we stop here.
2. This smaller N can be a prime factor or it can be another smaller number, we don't care, all we have to do is to repeat this process until $N=1$.

TEST YOUR PRIME FACTORS KNOWLEDGE

Solve UVa problems related with prime factors:

583 - Prime Factors

Prime Numbers

Prime numbers are important in Computer Science (For example: Cryptography) and finding prime numbers in a given interval is a "tedious" task. Usually, we are looking for big (very big) prime numbers therefore searching for a better prime numbers algorithms will never stop.

1. Standard prime testing

```
int is_prime(int n) {
    for (int i=2; i<=(int) sqrt(n); i++) if (n%i == 0) return 0;
    return 1;
}void main() {
    int i, count=0;
    for (i=1; i<10000; i++) count += is_prime(i);
    printf("Total of %d primes\n", count);
}
```