Chapter 1

Solutions

1.1 Multiples of 3 and 5

> If we list all the natural numbers below 10 that are multiples of 3 and 5, we get 3, 5, 6, and 9. The sum of these multiples is 23.
>
> Find the sum of all the multiples of 3 and 5 below 1000.

The first problem is pretty much straightforward. It asks us to find the sum of all multiples of 3 and 5, but any multiple must not exceed $1000 - 1 = 999$. But wait, is this task really that simple? The only trick is that when we calculate the multiples, some of the multiples of 3 and 5 are the same, e.g. 15, 30, ... That means, that if we do not implement this task in smart way, some of the multiples might get counted twice. Those numbers are exactly the multiples of the least common multiple of 3 and 5, which is 15. We denote that as $\operatorname{lcm}(3, 5) = 15$.

Let us take a look at naive solution:

```
#include<stdio.h>

int main()
{
    int sum = 0;
    for (int i = 1; i <= 333; i++)   // we iterate from 1 to (int)999/3 = 333
        sum += 3 * i;
    for (int i = 1; i <= 199; i++)   // we iterate from 1 to (int)999/5 = 199
        sum += 5 * i;
    for (int i = 1; i <= 66; i++)    // we iterate from 1 to (int)999/15 = 66
        sum -= 15 * i;               // we must subtract the twice counted numbers
    printf("%d\n", sum);

    return 0;
}
```

Note that if you implement this program, it will print the value instantly, therefore you might pose the question, why is this a naive solution? The reasons are two. Firstly, we use three for loops. That results in asymptotic complexity of $\mathcal{O}(m/x + m/y + m/\operatorname{lcm}(x, y))$, where x and y are the numbers whose multipliers we want to sum and m is the upper bound for all multiples. Now consider we are given extremely large numbers. Although the program runs in linear time,

it will take more and more time to get through those three loops. We can clearly see, that we could get rid of three loops and make only one loop, as in next implementation. Wait until you see the real reason why this implementation is not as beautiful as it could be (not to mention it is tiresomly slow - oh, wait, linear is slow, can you do it faster, letâĂŹs say logarithmic? Hell no, CONSTANT!).

```
#include<stdio.h>

int main()
{
    int sum = 0;
    for (int i = 3; i < 1000; i++)
    {
        if (i % 3 == 0 || i % 5 == 0)
            sum += i;
    }
    printf("%d\n", sum);

    return 0;
}
```

Note that this implementation does not require any subtraction of multiples of 15, because we check every integer between 3 and 999 inclusively only once. We also expect this program to run slower then the three for one, because in this case we have to loop through $m-2 = 998$ numbers and each time check whether that number is a multiple of 3 or 5, whereas in three for loop we only loop through $\lfloor (m-1)/x \rfloor + \lfloor (m-1)/y \rfloor + \lfloor (m-1)/\operatorname{lcm}(x,y) \rfloor = 333+199+66 = 598$ numbers and we do not need to do any checks.

The second reason, and here starts the real solution to the problem, is that what we are doing is actually computing a sum of three *arithmetic series*. An arithmetic series is a sum of numbers that are computed using relation

$$a_1; \quad a_i = a_1 + d(i-1),$$

where d denotes a difference or *distance* between two numbers in series. In our case we have three arithmetic series, first given by

$$a_1 = 3; \;\; a_i = 3 + 3(i-1) = 3i,$$

the second by

$$b_1 = 5; \;\; b_i = 5 + 5(i-1) = 5i$$

and the third by

$$c_1 = 15; \;\; c_i = 15 + 15(i-1) = 15i.$$

We are actually subtracting the sum of $\{c_i\}_{i=1}^{66}$ series. Now, beautiful mathematics gives us a formula for computing sum of arithmetic series. The formula is

$$\sum_{i=1}^{n} a_i = \frac{n}{2}(a_1 + a_n).$$

Since in our case $a_n = 3n$, $b_n = 5n$, and $c_n = 15n$, the above equation simplifies into

$$\sum_{i=1}^{n} a_i = a_1 \cdot \frac{n(n+1)}{2}.$$

You can simply make your own calculations to see that formula is indeed correct. Let us now implement the program WITHOUT any `for` loops at all! We use a formula derived above and get

$$\text{sum} = \sum_{i=1}^{333} 3i + \sum_{i=1}^{199} 5i - \sum_{i=1}^{66} 15i = 3 \cdot \frac{333 \cdot 334}{2} + 5 \cdot \frac{199 \cdot 200}{2} - 15 \cdot \frac{66 \cdot 67}{2}.$$

```
#include<stdio.h>
#include "campovski.h"              // needed for GCD function

int main()
{
    int x, y, m;                    // we let user input starting numbers
    scanf("%d %d %d, &x, &y, &m);   // (in our case "3 5 1000")
    m--;                            // number below m, m not included (1000-1)

    int z = x * y / gcd(x, y);      // LCM(x,y) = x * y / GCD(x,y)
    int n1 = (int) m / x;           // calculate range of sum for x (3)
    int n2 = (int) m / y;           // calculate range of sum for y (5)
    int n3 = (int) m / z;           // calculate range of sum for LCM(x,y) (15)

    int sum = x * n1 * (n1 + 1) / 2;
    sum += y * n2 * (n2 + 1) / 2;
    sum -= z * n3 * (n3 + 1) / 2;

    printf("%d\n", sum);

    return 0;
}
```

You might say taht you could do that faster with your own calculator. Well, the beauty of coding is finding mathematical background of given problem and optimizing the solution.

I tested all three of the above programs with number $x = 3$, $y = 5$, and $m \in \{10^3, 10^6, 10^9, 10^{12}\}$. The results are astounding (see table below). I also had to change `int`s to `long long`s.

Algorithm	$t(m = 10^3)$	$t(m = 10^6)$	$t(m = 10^9)$	$t(m = 10^{12})$
Three `for` loops	0.000002	0.001528	1.384336	1321.064965
One `for` loop	0.000004	0.003579	3.020462	
Formula	< 0.000001	< 0.000001	< 0.000001	< 0.000001

Table 1.1: Table shows running time (in seconds) of given algorithms in relation to m. Time was measured with help of `time.h`.

Note, hot lovely are the running times of first two algorithms increasing with m. Thousand times greater problem, thousand times more time needed. The third algorithm obviously runs in constant time.

1.2 Even Fibonacci numbers

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

$$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

Let us start with example from the task. The first ten terms of Fibonacci sequence, as the task states it, are 1, 2, 3, 5, 8, 13, 21, 34, 55, and 89. Even-valued terms, that do not exceed 34 are 2, 8, and 34. Therefore, the sum of the even-valued terms that do not exceed 34 is $2+8+34 = 44$.

Before we get into programming, let us first discuss how terms of Fibonacci sequence are calculated. The first two terms, according to task, are $F_1 = 1$ and $F_2 = 2$. (Mind, that some sources state Fibonacci sequence as 1, 1, 2, 3, 5, 8, 13,... or even 0, 1, 1, 2, 3, 5, 8, 13,...) The next term is acquired by adding previous two together. We end up with formula

$$F_n = F_{n-1} + F_{n-2};\ n \geq 3.$$

This is recursive formula with starting values of $F_1 = 1$ and $F_2 = 2$. At this point you might think that we are going to recursion in order to calculate terms of Fibonacci sequence, but we are not. The reason is, that if we want to compute n-th term of Fibonacci sequence, we can use recursion. In our case, we do not know what term is the first that exceeds 4000000, therefore we cannot use recursion. Besides, naive recursion (if we could use it) would calculate the same terms over and over again - see figure below. We could use a concept called *memoization* and get rid of this problem, but as I said, we cannot use this technique in this situation.

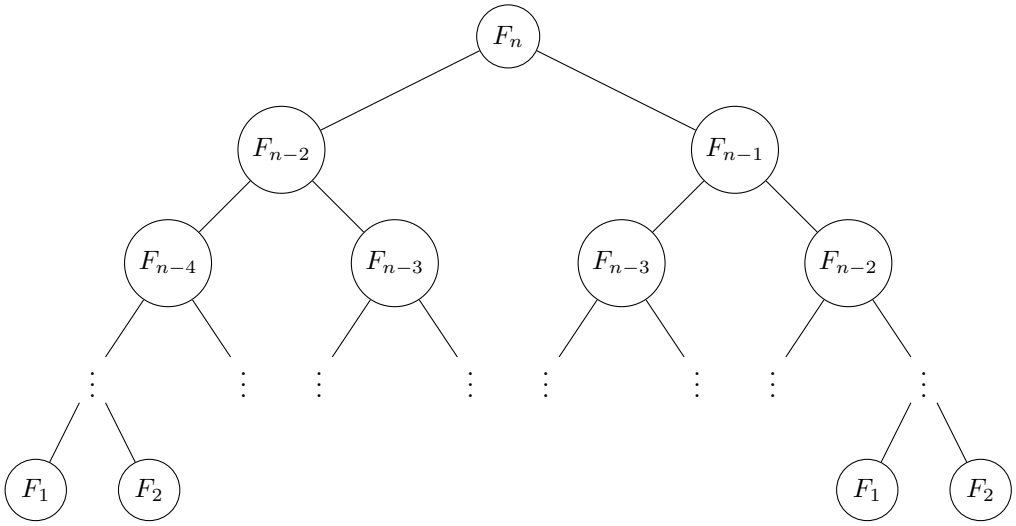

Figure 1.1: Recursion tree for Fibonacci sequence. Note, how terms keep repeating.

Since we cannot use recursion from n to 1, we are going to go from 1 to n, summing the previous two terms together to obtain the next one. We are going to check whether that term is even and if it is, we are going to add it to sum.

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

```
int main()
{
    int x = 1;              // x represents F(n-2)
    int y = 2;              // y represents F(n-1)
    int tmp;                // tmp represents F(n)
    int sum = 0;
    while (x <= 4000000)
    {
        if (!(x % 2))       // check if x is even
            sum += x;
        tmp = x + y;        // F(n) = F(n-1) + F(n-2)
        x = y;              // F(n-1) becomes F(n-2) for next iteration
        y = tmp;            // F(n) becomes F(n-1) for next iteration
    }
    printf("%d\n", sum);

    return 0;
}
```

There are probably few things I should explain. First is the condition in `if` clause. In C, as well as in some other languages, number 0 of type `int`, beside of being a regular number, stands for *false* (mind, that C does NOT have boolean variables). Thus, we do not need to make comparison of `0 == 0`, but we can simply write `!0`, which evaluates to `1`. Next thing is the way we compute terms for next iteration. First, we sum current terms and save the result in temporary variable `tmp`. Then we rewrite `x` and `y` with `y` and `tmp` respectively, thus shifting the sequence to the left. The asymptotic complexity of this algorithm is $\mathcal{O}(n)$, where n is the number which terms must not exceed - in our case $n = 4000000$.

Remember, that I said that some sources state Fibonacci sequence as 1, 1, 2, 3, 5, 8, 13, 21, 44,... Observing this sequence for long enough we can see, that even-valued terms are F_{3i} for all $i \in \mathbb{N}$. Proving this is simple. The sum of two odd numbers is even and the sum of even and odd number is odd. Looking at the sequence we can see that this proves the statement by law of induction. Therefore, we can be sure that if we add every third term of Fibonacci sequence to our final sum, we will obtain the correct result. The following algorithm does exactly that. Each iteration we compute the next three terms of Fibonacci sequence and always add the third to final sum.

```
#include<stdio.h>

int main()
{
    int x = 1;              // x represents F(n-2) - always odd
    int y = 1;              // y represents F(n-1) - always odd
    int z = 2;              // z represents F(n) - the even term
    int sum = 0;

    while (z <= 4000000)
    {
        sum += z;           // always add z since it is always even
        x = y + z;          // F(n+1) = F(n-1) + F(n) (x becomes F(n-2))
        y = z + x;          // F(n+2) = F(n) + F(n+1) (y becomes F(n-1))
        z = x + y;          // F(n+3) = F(n+2) + F(n+2) (z becomes F(n))
```

```
    }

    printf("%d\n", sum);

    return 0;
}
```

This algorithm still runs in $\mathcal{O}(n)$ but we are advancing three times faster than in previous algorithm, because we calculate three terms in each iteration of `while` loop and omitting the `if` clause, thus expecting this algorithm to run approximately three times faster does not sound stupid. The only problem is that we are also doing more summations, which will result in algorithm not being three times as fast as the previous one, but it should still be faster.

Now it is time for some maths. We already figured out that even-valued terms are F_{3i}. If we manage to rewrite the recusrion formula $F_n = F_{n-1} + F_{n-2}$ into $F_n = \alpha f_{n-3} + \beta F_{n-6}$, we will obtain a formula that uses only every third Fibonacci number and we will not have to bother with odd-valued terms. Let us try to do that.

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &= (F_{n-2} + F_{n-3}) + (F_{n-3} + F_{n-4}) \\ &= \big((F_{n-3} + F_{n-4}) + F_{n-3}\big) + \big(F_{n-3} + (F_{n-5} + F_{n-6})\big) \\ &= 3F_{n-3} + (F_{n-4} + F_{n-5}) + F_{n-6} \\ &= 4F_{n-3} + F_{n-6} \end{aligned}$$

Easy peasy lemon squizy! Based on formula derived, we can implement next program.

```
#include<stdio.h>

int main()
{
    int x = 2;              // x represents F(n-6)
    int y = 8;              // y represents F(n-3)
    int tmp;                // tmp represents F(n)
    int sum = 0;

    while (x <= 4000000)
    {
        sum += x;
        tmp = 4 * y + x;    // F(n) = 4 * F(n-3) + F(n-6)
        x = y;              // F(n-3) becomes F(n-6) for next iteration
        y = tmp;            // F(n) becomes F(n-3) for next iteration
    }

    printf("%d\n", sum);

    return 0;
}
```

Can you guess, how much faster, if faster, will this algorithm run compared to the second one? It still runs in $\mathcal{O}(n)$, we still advance by three terms each iteration... But hey, did you forget about the number of operations again?! I hope not. This time, we do one multiplication more than in second algorithm, but two summations less. Therefore, we expect this program

to run faster than the second one, and therefore also faster than the first one. Actually, if we compare the first program to the last one, we can see, that the difference is one `if` clause more in the first one and one small multiplication more in the third one. We might actually hope that the last algorithm will run three times as fast as the first one.

I ran all three programs and measured the time with `time.h` but the precision of timing was too low to see much difference. The first program finished in average time of 0.000002 seconds, and the other two finished in under 0.000001 seconds. with $n = 9\,000\,000\,000\,000\,000\,000 \approx$ `LLONG_MAX` (9 quintillion). I decided to make each algorithm run more times. The following table shows the results.

Algorithm	$t(r = 10^3)$	$t(r = 10^6)$	$t(r = 10^9)$
`for` and `if`	0.001498	0.335005	330.163120
Naive three step	0.000784	0.183093	181.040890
Three step formula	0.000215	0.102709	97.865098

Table 1.2: Table shows the running time (in seconds) of given algorithms in relation to r, where r is the number of repetitions the core of program (the algorithm) ran. Time was measured with help of `time.h`.

The results are just like we expected them to be. The *three step formula* (last algorithm) is three times as fast as the *`for` and `if`* (first algorithm). The *naive three step* (second algorithm) is somewhere in the middle.

1.3 Largest prime factor

> The prime factors of 13195 are 5, 7, 13, and 29.
>
> What is the largest prime factor of the number 600851475143?

There is a pretty straightforward solution to this problem that seemingly runs in $\mathcal{O}(n)$. What we can do is to loop through every number below 600851475143 and check whether it is a prime and whether it divides 600851475143. The most easily grasped solution with high running time. But why did I say it *seemingly* runs in $\mathcal{O}(n)$. The answer is simple. We have to loop through n numbers, for each check, whether it is a prime or not, that adds another n in game. The total running time is then $\mathcal{O}(n^2)$. With $n = 600851475143$, my head starts burning. I just hope my computer will not. Let us implement this monstrous piece. I advise you not to try this at home. (Not sure should I do a backup first or not...)

```
#include<stdio.h>
#include<stdlib.h>
#define N 600851475143

int is_prime(long n);

int main()
{
    long largest = 1;
    for (long i = 2; i <= N; i++)
    {
        if (N % i == 0 && is_prime(i))
```

```
            largest = i;
    }

    printf("%li\n", largest);

    return 0;
}

int is_prime(long n)
{
    for (int i = 2; i < n; i++)
    {
        if (n % i == 0)
            return 0;
    }
    return 1;
}
```

Mind two things: Although this is a not do at home algorithm, this might come in handy sometime in the future. Mind how we wrote `N % i == 0` before `is_prime(i)` in `if` clause of function `main`. Since we wrote it this way, the computer will first check whether `i` divides `N`. If not, it will not bother executing `is_prime(i)`, since the `AND` clause will evaluate to `0` no matter the second argument. By doing that, we make sure, that we do not check every number for primality and therefore the running time of algorithm is reduced substantially. What we could also do is return n when the for loop comes to it and `largest == 1`, since we did not find any other divisor of n, meaning n is prime and we do not need to check its primality.

The second thing you should mind is that we do not need to check whether the new prime divisor is greater than the largest. You might have seen it by yourself why is so, but if not, here is the deal. The `i`, which is a possible prime divisor, increases and never decreases. Therefore, it always holds `i > largest`.

There are few things we could do better in implementation above and we would not even need to think hard. The first that comes to my mind is, that when we check for primality, we do not need to loop from 2 to n. We only need to loop to $\sqrt{n}$. The reason is that if a number has a higher divisor then its square root, the result of division is smaller than number's square root and it is also a divisor. Therefore we can only check in range from 2 to $\sqrt{n}$.

It is time for some maths. You have learned about *sieve of Eratosthenes* in primary school for sure. The idea behind the sieve is to generate an array of numbers between 2 and n. Then we start and cross out all multiples of 2, except 2, which is prime. After that, we cross out all multiples of next number that is not crossed, that is 3. We continue by crossing the multiples of all numbers that are not crossed. By doing so, we end up with table of prime numbers. For implementation of Eratosthenes' sieve see appendix. The only problem is, that we cannot use it here, because we cannot store 600851475143 bytes anywhere. We will have to dig further, but here is the implementation using sieve of Eratosthenes. It works for small numbers and it works fast.

```
#include<stdio.h>
#include "campovski.h"                  // needed for ERATOSTHENES
#define N 100003                        // some small number

int main()
```

```
{
    char* sieve = eratosthenes(N);      // get prime array
    int largest = 1;

    for (int i = 2; i <= N; i++)
    {
        if (!sieve[i] && N % i == 0)    // if i is prime, sieve[i] = 0
            largest = i;
    }

    printf("%d\n", largest);

    return 0;
}
```

The idea for next algorithm comes directly from Eratosthenes but instead of creating a sieve, we completely divide out each of the divisor of n. By doing that we make sure that the next divisor will certainly be prime and therefore we do not need to check for divisor whether it is prime or not. By doing so we spare a lot of time, because every time we find a divisor a, we divide n by a as many times as we can, meaning we end up with smaller and smaller n, thus the calculation become faster and faster. Next implementation shows how that is done.

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

int main()
{
    long long n = 600851475143;
    long long factor = 2;
    long long largest;

    while (n > 1)
    {
        if (!(n % factor))
        {
            largest = factor;
            do {
                n /= factor;
            } while (!(n % factor));
        }

        factor++;
    }

    printf("%lli\n", largest);

    return 0;
}
```

This algorithm can be improved in two ways. It is obvious that 2 is the only even-valued prime number. Therefore we can treat it on its own. By doing so, we can increase `factor` by 2 on each iteration and only traverse over odd numbers. The second improvement comes with

realisation that every number can have at most one prime factor greater than $\sqrt{n}$. That leaves us with two options, either we found a factor lower then $\sqrt{n}$, divided it out and continued with next iteration, or we came to $\sqrt{n}$ and did not find a factor. In second option, we can stop checking since if there is no prime smaller then n that would divide n, then the only numbers that divide n are 1 and n. That means n is a prime and we return it. The next implementation does exactly that.

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

int main()
{
    long long n = 600851475143;
    long long largest = 1;

    if (!(n % 2))
    {
        largest = 2;
        do {
            n /= 2;
        } while (!(n % 2));
    }

    long long factor = 3;
    long long max_factor = (long long) sqrt(n);

    while (n > 1 && factor <= max_factor)
    {
        if (!(n % factor))
        {
            largest = factor;
            do {
                n /= factor;
            } while (!(n % factor));
            max_factor = (long long) sqrt(n);
        }

        factor += 2;
    }

    if (n == 1)
        printf("%lli\n", largest);
    else printf("%lli\n", n);

    return 0;
}
```

I measured the running time of algorithms and the results were quite shocking. The first algorithm needed..., while the second and third one were surprisingly fast. The next table shows running times of algorithms with respect to n. Each program was run 1000 times to ensure smaller relative error, except for *one step* `factor` on $n = 100123456789$.

As you can see, running times are quite strange. For small numbers that are not prime the `one step factor` is faster than `two step + sqrt`, because square root takes too much time

Algorithm	$t(n = 10^3)$	$t(n = 10^6)$	$t(n = 10^5 + 3)$	$t(n = n_1)$	$t(n = n_2)$
Monster	0.012390	8.448938	1.754996		
Sieve	0.009984	3.659880	0.454618	–	–
One step `factor`	0.000478	0.000634	0.214148	0.130969	$\sim$ 116 days
Two step + `sqrt`	0.006801	0.007498	0.001555	0.023457	1.362939

Table 1.3: Table shows the running time (in seconds) of given algorithms with respect to n, where n is the number whose greatest prime factor we are trying to calculate. $n_1 = 600851475143$ and $n_2 = 100123456789$ (n_2 is prime). Time was measured with help of `time.h`.

to evaluate. On the other hand, if n happens to be prime, the `one step factor` takes a lot of time, because it has to search through all numbers below n before realizing it could actually return an answer. Therefore, one can clearly see that *two step + `sqrt`* is a way to go. The *sieve* is quite fast, but not appropriate for this problem, since has such a specific implementation that it simply cannot compete. We included it just for broadening mind. Mind also, that we computed sieve only once and ran the rest of the program 1000 times. Places with – denote we could not allocate such a huge sieve and empty space means we did not run the program because it would take too long to execute.

1.25 1000-digit Fibonacci number

The Fibonacci sequence is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}, \text{ where } F_1 = 1 \text{ and } F_2 = 1.$$

Hence the first 12 terms will be:

$$\begin{aligned} F_1 &= 1 \\ F_2 &= 1 \\ F_3 &= 2 \\ F_4 &= 3 \\ F_5 &= 5 \\ F_6 &= 6 \\ F_7 &= 13 \\ F_8 &= 21 \\ F_9 &= 34 \\ F_{10} &= 55 \\ F_{11} &= 89 \\ F_{12} &= 144 \end{aligned}$$

The 12th term, F_{12}, is the first term to contain three digits.

What is the index of the first term in the Fibonacci sequence to contain 1000 digits?

This is one of the problems that we are particularly not lucky to be programming in C. We cannot store the 1000 digit number in any of number types. We will have to use array of `int`s (we could use `char`s, but it is really not necessary, since we do not need to spare with memory). The only possible solution to this problem is brute force, since there are no known rules for how

Fibonacci numbers' lengths increase. In languages like Python we could write the solution to this problem very easily because we do not need to worry about storing such huge numbers. The solution in Python is this:

```
x = 1
y = 1
count = 2
while len(str(y)) != 1000:
    x, y = y, x + y
    count += 1
print count
```

Simple as f***. Now, the solution in C is not much harder, we just have to be careful with arrays. Since we will not care about memory, we can allocate three 1000 `int`s long arrays, one for F_{n-1}, one for F_n, and one for F_{n+1}. Then we loop over all the digits in F_{n-1} and F_n and calculate each digit of F_{n+1} by piece. The only problem is, that we must take care of the situation where `F(n-1)[i] + F(n)[i] >= 10`. In this case, we have to remember to increment the next digit. Fairly simple, but quite painful. The next implementation does exactly that.

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

int main()
{
    int* x = (int*) malloc(1000*sizeof(int));
    int* y = (int*) malloc(1000*sizeof(int));
    int* tmp = (int*) malloc(1000*sizeof(int));
    int inc = 0;                                    // increment on next step

    x[0] = 1;                                       // F(1) = 1
    y[0] = 1;                                       // F(2) = 1
    int count = 2;                                  // we already know 2 numbers

    while (!y[999])                                 // loop while 1000th digit is 0
    {
        for (int i = 0; i < 1000; i++)
        {
            int s = x[i] + y[i] + inc;              // sum of i-th digit + increment
            tmp[i] = s % 10;                        // digit of F(n+1)
            x[i] = y[i];                            // F(n) becomes F(n-1)
            y[i] = tmp[i];                          // F(n+1) becomes F(n)
            inc = (s >= 10) ? 1 : 0;                // if s>=10, we must inc on next iter
        }
        count++;
    }

    printf("%d\n", count);

    return 0;
}
```

Now if we would want to spare with memory, we would have two options. The first thing we could do is change all `int` values to `short`s. That would spare a lot of the memory we used for

variables. But furthermore, since we know that we will only be using really small numbers at a time (less then $9+9+1 = 19$), we could allocate a `char` array and spare even more. Many people think that `char` stands for *character*. Well, it actually does, but the `char` type can normally store integral values, just as long as they do not exceed 2^8. Instead of 4 bytes (can store up to 2^{32}) for an `int`, we could only use 1 byte (can store up to 2^8) for a `char`. The implementation is the same.

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

int main()
{
    char* x = (char*) malloc(1000*sizeof(char));
    char* y = (char*) malloc(1000*sizeof(char));
    char* tmp = (char*) malloc(1000*sizeof(char));
    char inc = 0;

    x[0] = 1;
    y[0] = 1;
    int count = 2;

    while (!y[999])
    {
        for (int i = 0; i < 1000; i++)
        {
            char s = x[i] + y[i] + inc;
            tmp[i] = s % 10;
            x[i] = y[i];
            y[i] = tmp[i];
            inc = s >= 10 ? 1 : 0;
        }
        count++;
    }

    printf("%d\n", count);

    return 0;
}
```

Let me take another note on `char`s. One `char` is 1 byte, 1 byte is 8 bits and 1 bit can store two values, `0` and `1`. Since we can store 8 bits in a byte, that means, we can store 2^8 values in 1 byte. Therefore, if we only have less than $2^8 = 256$ different signs to store, we can use a `char`. We use so called ASCII table, which is nothing else but a bijection between symbols like characters, numbers, punctuation, etc. and numbers between 0 and 255. For example, a character `"A"` has an ASCII value of `65`. Therefore, if we write something like `char c = "A";`, what actually gets stored in `c` is the number `65`, which represents `"A"`. That means, that if we store `42` in `char`, the computer might understand that as a character with ASCII value of 42, that is `*`, but it can serve as a pure number for our needs, see what I mean? For full ASCII table go ahead and google it, there is really no point in creating one here.

Also, we will not be making any time comparison here, since we only have one algorithm, the running time of which is clearly $\mathcal{O}(n)$, where n is the length of Fibonacci number, in our case $n = 1000$.