5 Fold, Scan, State

Key concept:

1. The fold and scan functions are used to keep track of states.
    - It can be considered as eliminating a lot of intermediate steps, where the number of intermediate steps may change based on the length of the list.
    - It is somewhat similar to using a mutable state, but less things to keep track of.

5.1 `List.fold`

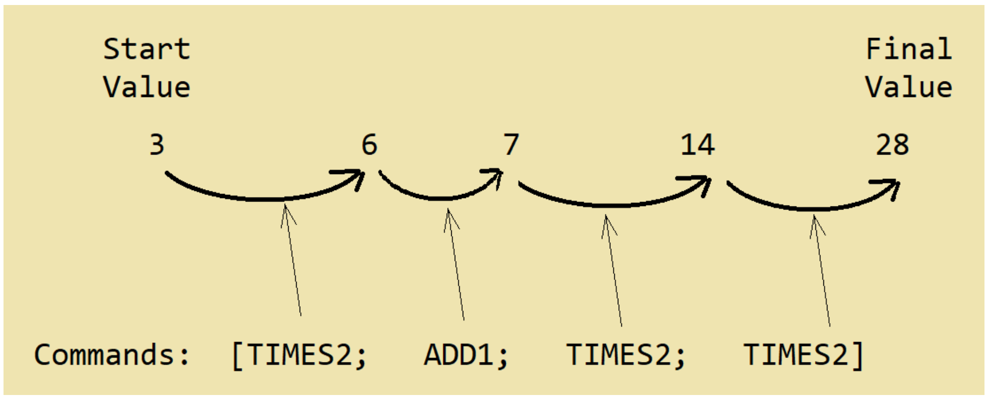
Let us look at an example:

```
let listOfCommands1 =
    ["TIMES2"; "ADD1"; "TIMES2"; "TIMES2"]

let ChangingFunction prevResult command =
    if command = "TIMES2" then prevResult * 2
    else if command = "ADD1" then prevResult + 1
    else failwith "Unknown operation."

let startingValue = 3
```

Here, `ChangingFunction` tells us how to modify a value based on the Command `string` accepted.

Remark: We used the `failwith` to throw an exception/crash the program when we encounter an unknown operation. There is a better way to handle this situation, but for the purpose of this talk we will ignore it.

```
let result1 =
    List.fold ChangingFunction startValue listOfCommands1
// val result1 : int = 28
```



Here, we have a `startValue` of 3, and we go through the `listOfCommands1` and evolve the `startValue` based on the `ChangingFunction`.

An equivalent implementation would be the following:

```
let result1_version2 =
    let intermediateResult1 =
        ChangingFunction startValue listOfCommands1.[0]

    let intermediateResult2 =
        ChangingFunction intermediateResult1
            listOfCommands1.[1]

    let intermediateResult3 =
        ChangingFunction intermediateResult2
            listOfCommands1.[2]

    let finalResult =
        ChangingFunction intermediateResult3
            listOfCommands1.[3]

    finalResult
```

Or if we use mutable, then:

```
let result1_version3 =
    let mutable valueSoFar = startValue
    for command in listOfCommands1 do
        let updatedValue =
            ChangingFunction valueSoFar command
        valueSoFar <- updatedValue
    // return
    valueSoFar
```

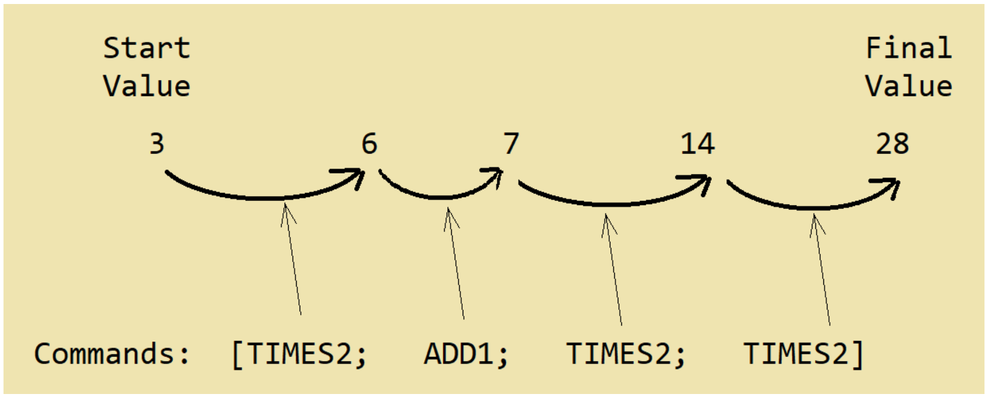
This is most similar to codes that you may write in Java/C++

Warning: If you use VisualStudio /VisualStudioCode, you may see that `valueSoFar` is highlighted yellow in your editor, as a warning that there is a mutable value in our program. As mentioned before, F# discourages the usage of mutable values.

5.2 Dependance on Starting and Intermediate Value

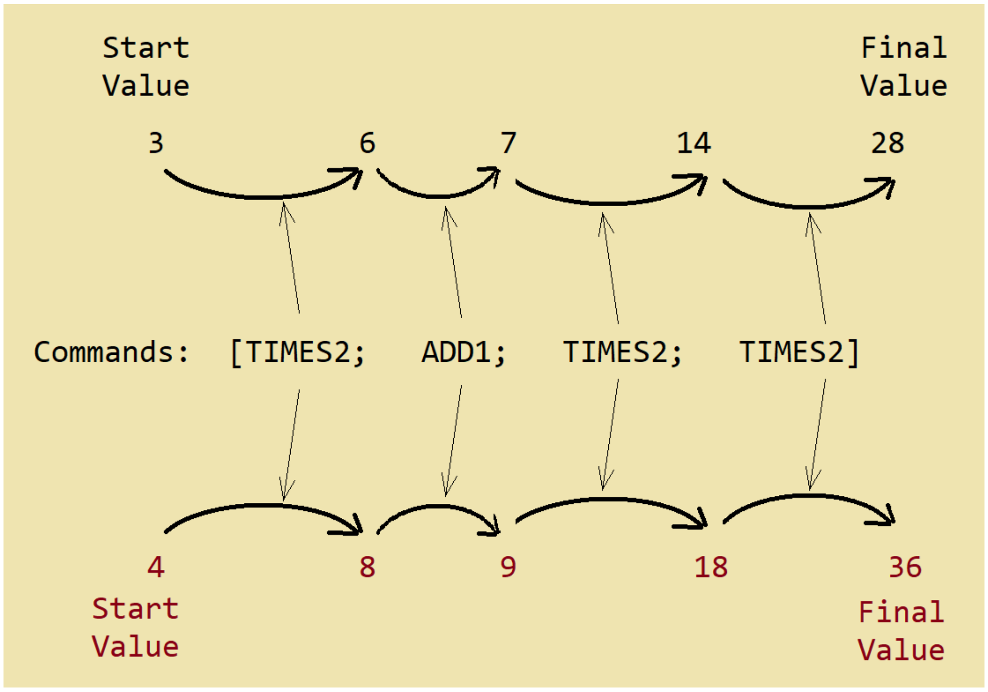
Dependance on Starting Value

```
let result1 =
    List.fold ChangingFunction startValue listOfCommands1
// val result1 : int = 28
```



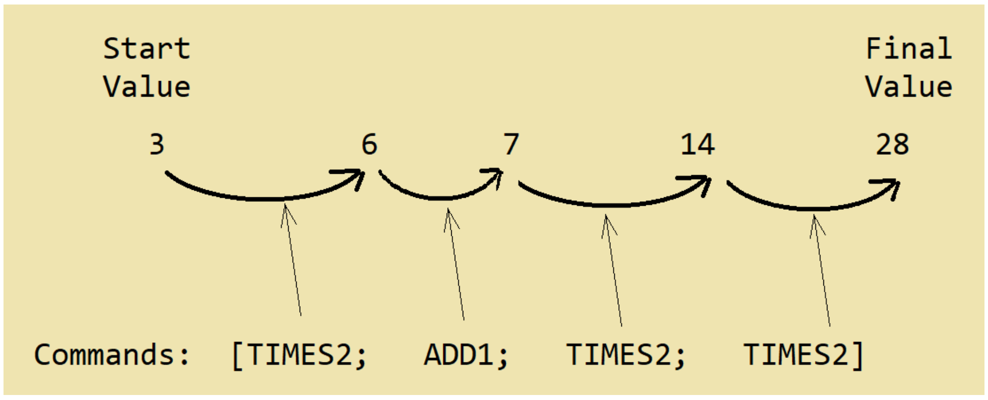
Notice that the folding process depends on the starting value:

```
let startValue2 = 4
let result2 =
    List.fold ChangingFunction startValue2 listOfCommands1
// val result2 : int = 36
```



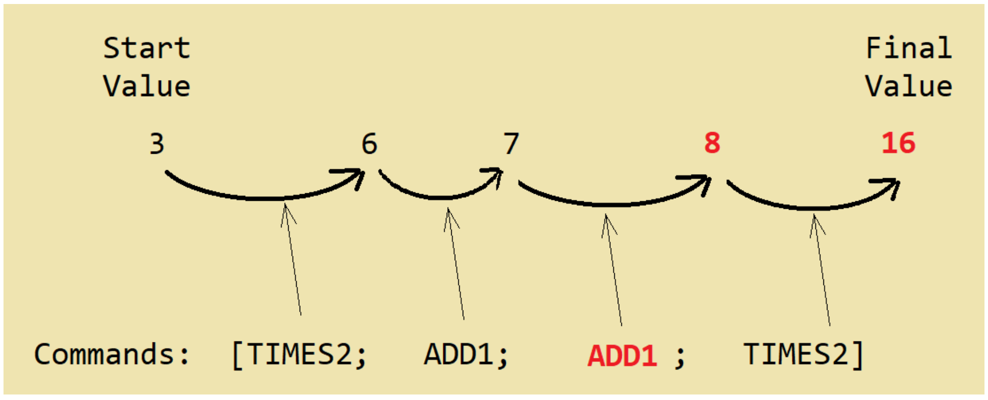
Dependance on Intermediate Value

```
let result1 =
    List.fold ChangingFunction startValue listOfCommands1
// val result1 : int = 28
```



The folding process also depends on the intermediate values:

```
let listOfCommands2 =
    [TIMES2; ADD1; ADD1; TIMES2]

// startValue = 3

let result3 =
    List.fold ChangingFunction startValue listOfCommands2
```



5.3 Examples

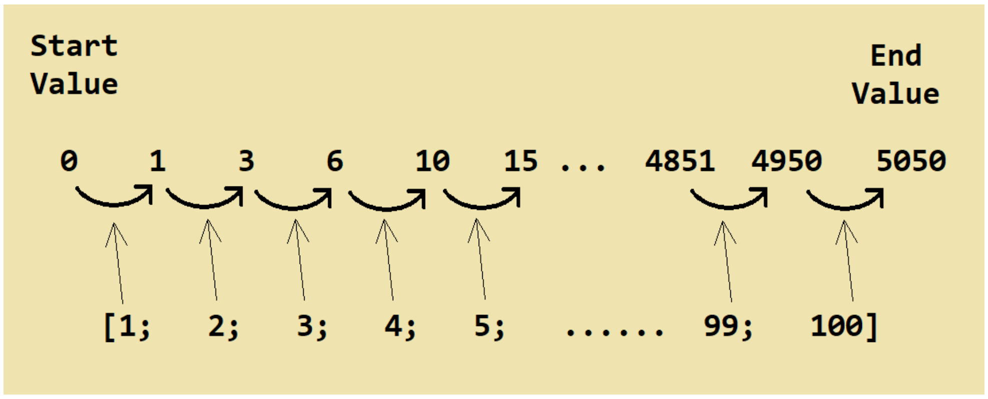
Sum a List

In order to sum a list, we can just use `List.sum`

```
let result4 = List.sum [1 .. 100]
// val result4 : int = 5050
```

Alternatively, we can imagine that we are adding up the value one by one, with a starting value of 0.

```
let result5 =
    [1 .. 100]
    |> List.fold (fun acc y -> acc + y) 0
// val result5 : int = 5050
```



Notice that the "folding function" is an anonymous/lambda function that accepts two variables:

- `acc`: The "accumulator" or intermediate result that is used to accumulate the informations.
- `y`: The elements from the list.

And the result of the "folding function" is the updated "accumulator" that will be passed on to the next accumulation stage.

If we are working in the previous case (where the accumulator is an integer, and the element of the list are strings `"ADD1", "TIMES2"`), then it is very clear which variable is which in the anonymous function.

However, in this case of re-implementing `List.sum`, both the accumulator and the element of the list are `int`, and so we may need to be careful when we are using a non-symmetric operator, i.e:

$$a + b = b + a \qquad a - b \neq b - a$$

Product of a List

Question. *Write a function that takes in an integer list, and outputs the product of all elements in that list (assume no integer overflow).*

When you use `List.fold`, consider two things:

1. Which starting value should you use?

2. What does your accumulator function do?

```
let ListProduct xList =
    xList
    |> List.fold (fun acc y -> ......)  .......

// What accumulating/folding function do you want to use?
// Which starting value should you use?
```

```
let result6 = ListProduct [1 .. 5]
// Expected Result: 1 x 2 x 3 x 4 x 5 = 120

let result7 = ListProduct [2; 3; 5; 7; 11; 13]
// Expected Result: 2 x 3 x 5 x 7 x 11 x 13 = 30030
```

Application: Euler Project Question 8

`https://projecteuler.net/problem=8`

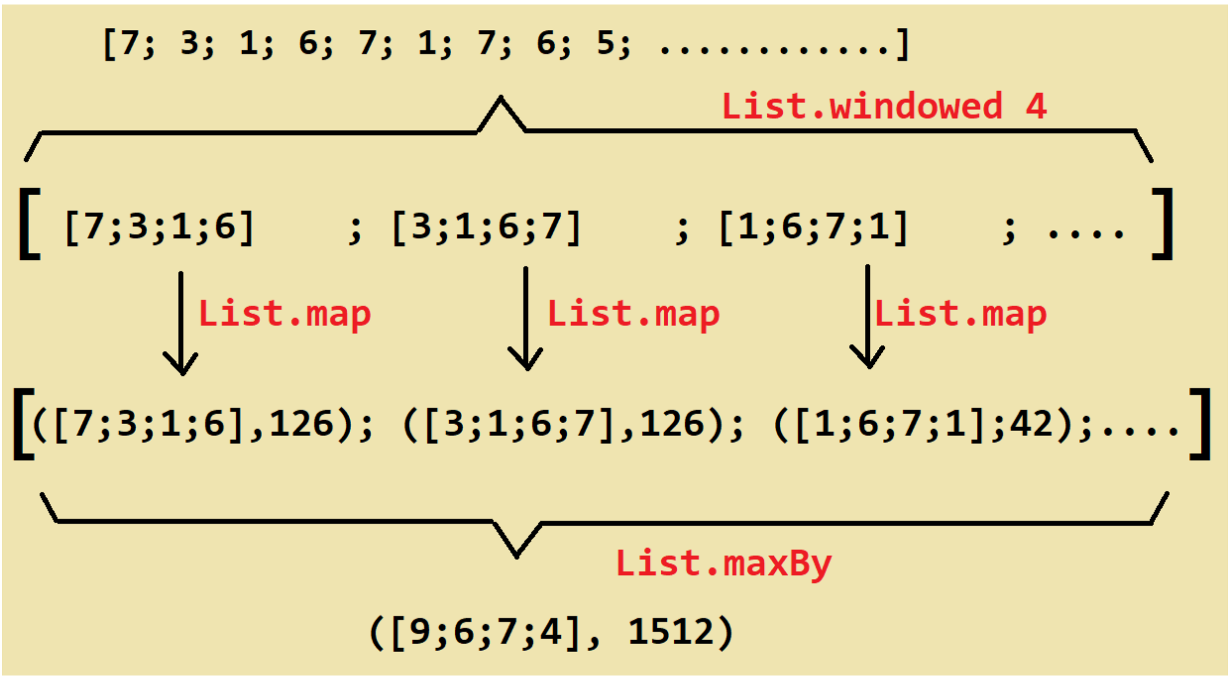
Modified Question. *Given a list of digits, find four adjacent digits with the largest product. For example, in the following number:*

$$7316717653133062491922511\mathbf{9674}4265747423553491949 34$$

The 4 consecutive digits that gives the largest product is $9 \times 6 \times 7 \times 4 = 1512$
(Notice that this line is the first line in the original question)

```
let digitList =
    [7;3;1;6;7;1;7;6;5;3;1;3;3;0;6;2;4;9;1;......]

let result8 =
    digitList
    |> List.windowed 4
    |> List.map (fun x -> x, ListProduct x)
    |> List.maxBy (fun (_,product) -> product)
// val result8 : int list * int = ([9;6;7;4], 1512)
```

We will use the picture below to illustrate what we are trying to achieve here.



Our goal here is to show that the `ListProduct` that we have implemented before using `List.fold` can be very powerful when combined with other `List` functions (e.g. `List.windowed`, `List.maxBy`, etc.) . To see how we approach the original Euler Problem, see the appendix.

Example: GCD of a list of integers

GCD for two variables already provided

You are given the following recursive `rec` function, that helps to calculate the greatest common divisor (GCD) of two integers. (This is Euclidean Algorithm)

```
let rec gcd x y =
    if x < 0 || y < 0 then failwith "cannot accept negative numbers"
    if x > y then gcd y x
    else if x = 0 then y
    else gcd (y % x) x
```

Reminder: You do not need to re-implement this function. You can just use it.

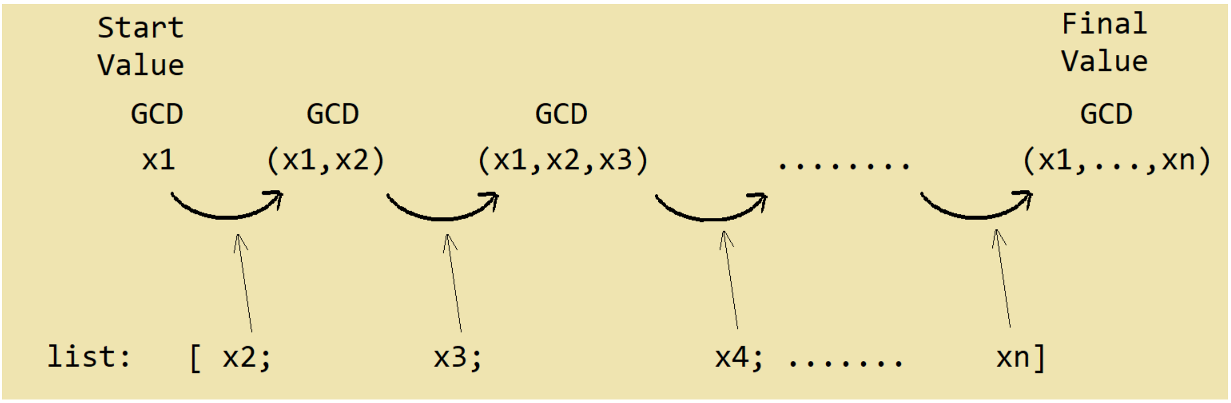
Question. *Given a list of (positive) integers, find the greatest common divisor (GCD) of those integers.*

Strategy: We make the following observation:

$$\begin{aligned} GCD\left(x_1, x_2, x_3\right) &= GCD\left[GCD\left(x_1, x_2\right), x_3\right] \\ GCD\left(x_1, x_2, x_3, x_4\right) &= GCD\left[GCD\left(x_1, x_2, x_3\right), x_4\right] \\ \vdots \quad &= \quad \vdots \\ GCD\left(x_1, x_2, \ldots, x_n\right) &= GCD\left[GCD\left(x_1, \ldots, x_{n-1}\right), x_n\right] \end{aligned}$$

```
let gcdOfList xList =
    let first = xList |> List.head
    let remaining = xList |> List.tail

    remaining
    |> List.fold gcd first
```

We will use the picture below to illustrate what we are trying to achieve here.



Exercise: Euler Project Question 5

`https://projecteuler.net/problem=5`

Modified Question. *Given a list of integers, find the lowest common multiple (LCM) of all those numbers. (Assume no integer overflow)*

Strategy: We make the following observation:

$$\begin{aligned}
LCM(x_1, x_2) &= LCM[LCM(x_1), x_2] \\
LCM(x_1, x_2, x_3) &= LCM[LCM(x_1, x_2), x_3] \\
LCM(x_1, x_2, x_3, x_4) &= LCM[LCM(x_1, x_2, x_3), x_4] \\
\vdots &= \vdots \\
LCM(x_1, x_2, \ldots, x_n) &= LCM[LCM(x_1, \ldots, x_{n-1}), x_n]
\end{aligned}$$

Hint: You can directly use the LCM function as your folding function. You do not need to re-implement it.

```
let lcm a b =
    a * b / (gcd a b)
```

So, we do not need to worry about our folding function, and we just need to worry about the starting value.

```
let lcmOfList xList =
    xList
    |> List.fold lcm ........

let result11 = lcmOfList [1 .. 10]
// Result: 2520

let result12 = lcmOfList [2;3;4;6;8;12]
// Result: 24
```

Warning: If you try to use this function on the list `[1 .. 20]`, you may either see an error, or see the following wrong result:

```
let result11 = lcmOfList [1 .. 20]
// Wrong Result: 18044195
// WRONG RESULT!!!!!!
```

Again, this is because of integer overflow (`int` cannot handle large numbers). You can see the Appendix on how to handle this situation.

Example: Iterate n-times

Take a look at the following code:

```
let result13 =
    [1 .. 20]
    |> List.fold (fun acc _ -> acc - 1) 100
// Answer: 80
```

Notice the following:

- The code above is similar to the following:

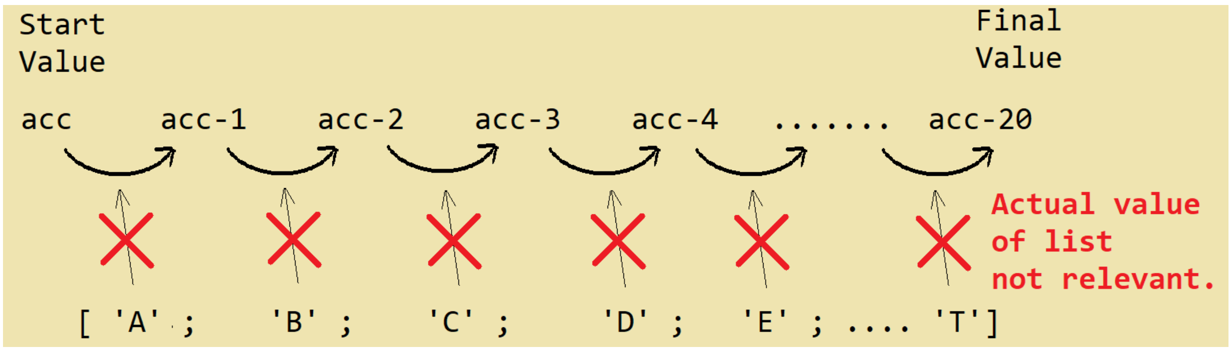
  ```
  let mutable resultMutable = 100
  for i in [1 .. 20] do
      resultMutable <- resultMutable - 1
  let finalResult = resultMutable
  // Answer: 80
  ```

  Notice that the `i` integer value is not used to modify the `resultMutable`, but rather is merely used to keep track of how many times we have applied the minus process.

- In the folding function, we used the underscore "_" to represent some variable that we do not intend to use. This means that the changing of the accumulator is irrelevant to the actual elements in the list. And so, we could have replaced that list with another different list with 20 elements, e.g.

  ```
  let result13_repeat =
      ['A' .. 'T']
      |> List.fold (fun acc _ -> acc - 1) 100
  // Answer: 80
  ```

  And in terms of diagram, this looks something like this:

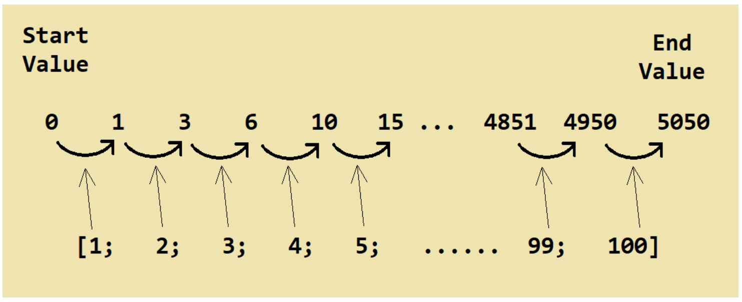

Start Value
Final Value
acc
acc-1
acc-2
acc-3
acc-4
....
acc-20
Actual value of list not relevant.
['A' ; 'B' ; 'C' ; 'D' ; 'E' ; .... 'T']

5.4 List.scan

List.scan is similar to List.fold, except that instead of returning the final result of the folding process, it returns ALL intermediate steps, final result, and initial value of the folding process.

For example, if you sum a list using List.fold:

```
let result14 =
    [1 .. 100]
    |> List.fold (fun acc y -> acc + y) 0
// val result14 : int = 5050
let result15 =
    [1 .. 100]
    |> List.scan (fun acc y -> acc + y) 0
// val result15 :
// int list = [0; 1; 3; 6; 10; ......; 4950; 5050]
```



Example: Fibonacci Numbers

The Fibonacci sequence (starting with 1 and 2) looks something like:

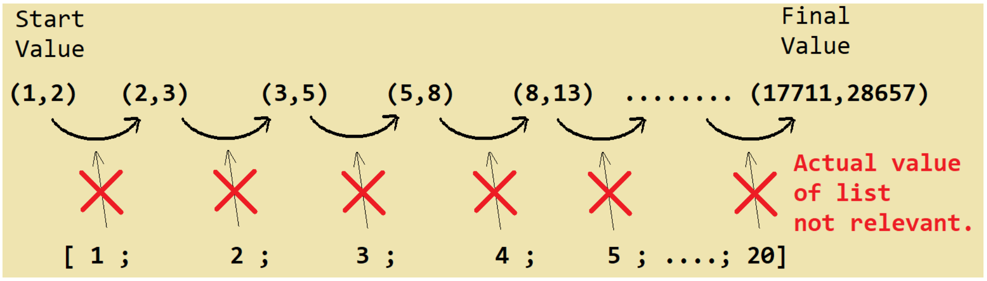
$$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

(For example, $1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8$, etc.)

We can generate Fibonacci numbers using the following code:

```
let resul16 =
    [1 .. 20]
    |> List.scan (fun (x,y) _ -> (y,x+y)) (1,2)
// val resul16 : (int * int) list = [(1, 2); (2, 3);
// (3, 5); (5, 8); (8, 13); (13, 21); ......];
```

In terms of diagram, this looks something like this:



Example: Euler Project Question 2

Original Question. *Find the sum of all even-valued fibonacci numbers below 4 million.*

1. We will first test whether the 41st fibonacci number exceeds four million or not.

   ```
   let first40FibNumbers =
       [1 .. 40]
       |> List.scan (fun (x,y) _ -> (y, x + y)) (1,2)
   // Result: [(1,2); (2,3); ......; (267914296, 433494437)]
   ```

   Since the 40th or 41st Fibonacci number already exceed our goal of four million (In fact, we do not even need to consider beyond the 32th number), working with 40 numbers is good enough.

2. Sum all even-valued fibonacci numbers below 4 million.

   ```
   let fibSum =
       [1 .. 40]
       |> List.scan (fun (x,y) _ -> (y, x + y)) (1,2)
       |> List.map (fun (x,y) -> x)
       |> List.filter (fun x -> x % 2 = 0)
       |> List.filter (fun x -> x < 4000000)
       |> List.sum
   // Result: 4613732
   ```

Exercise: Generate Tri-fibonacci numbers

Question. *Write some code that generates the tri-fibonacci numbers, i.e.*

$$1, 1, 1, 3, 5, 9, 17, 31, 57, \ldots$$

(For example: $1+1+1=3,\ \ 1+1+3=5,\ \ 1+3+5=9,\ \ 3+5+9=17,$ *etc.)*