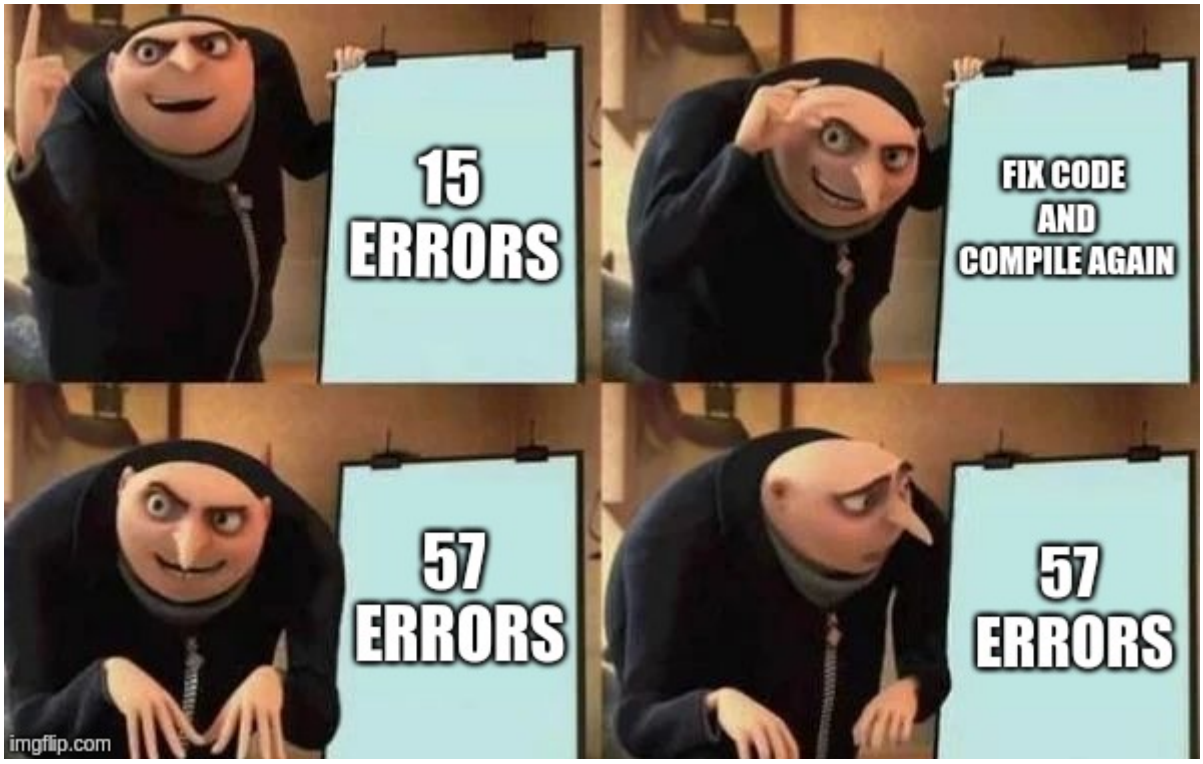
VG101 (22SU) Lab 3 Worksheet

Deadlines

The deadlines for basic and advanced exercises as well as demo project are all **Friday 23:59**. Late submissions will not be accepted. Good Luck!

Note: This time we extend the opening time of Lab3 on JOJ to next Monday, although you won't get any points for submissions after Friday. The purpose of this is to let you focus on completing the basic problems to get full marks before the deadline, and you may leave the more challenging problems to the weekend and test your answers on JOJ (if you are interested in these problems).

Skill: Debugging



(Image credit: This reddit thread)

Bugs are the largest and most common enemies of programmers. Everyone makes mistakes when coding, and these mistakes lead to crashes or strange behaviors of your programs, a.k.a. bugs.

Some interesting statistics:

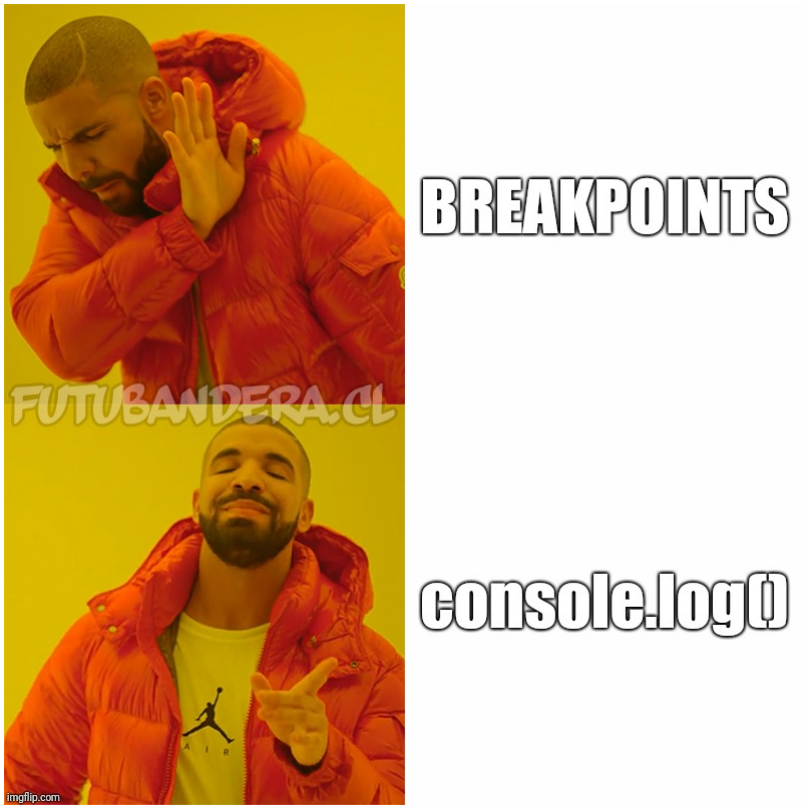
1. On average, a developer creates **70** bugs per 1000 lines of code.
2. Fixing a bug takes **30** times longer than writing a line of code.
3. **75%** of a developer's time is spent on debugging.

Some bugs are easy to discover and fix, such as a missing colon (`;`), while some are not as easy. Bugs that crash your program generally cause less trouble to deal with, but finding out why your program generates strange and unexpected output can take a lot of pain.

The process of trying to fix bugs is called ***debugging***. To debug a program, you first need to confirm that it is not functioning correctly. This is called ***testing***. Testing is hard! In large software companies, there are dedicated Quality Assurance Engineers whose job is to feed all kinds of strange and unusual input to their software and try hard to make the software crash or give incorrect output. Due to time limit, testing will not be covered in VG101 labs. You will learn more about it in VE280.

After you notice a bug, there are mainly two ways to deal with it:

1. Make your program output debugging information in intermediate steps of calculations. This is nicknamed "the `console.log()` method" among web developers.
2. Use a ***debugger*** to set ***breakpoints***, monitor the values of related variables, and keep trace of the ***function call stack***. Some of the debuggers have to be manually configured, but MATLAB provides a built-in debugger that is fairly easy to use.



(Image credit: This reddit thread)

It is meaningless to say which one is better. In production, almost all commonly-used software has built-in *verbose mode* that can generate debugging info, as well as ability to generate *logs*. The product manager will ask *bug reporters*, or users who want to report bugs they encounter when using the software, to provide these logs and verbose-mode output to help the developers figure out what was wrong. "The `console.log()` method" is also more easy to get started with (as is illustrated in the meme above). On the other hand, for developer themselves, a debuggers is often more easy to use in the long run, as it can pause the program and give the developer more freedom to explore the status of the program. It can also reveal more information to the developer.

Last but not least, a piece of advice from Salty Fish: **Avoid coding overnight!** Otherwise you will produce much more bugs than usual, and you will not be able to debug efficiently. Take a walk when you find yourself driven crazy by a bug that has been bothering you for hours. Get some sleep if you are tired. Debugging is hard, so do not make it harder by overloading your brain :)

Small Exercise about Debugging (8 marks)

Given the sample code below, use either breakpoint or `console.log()` method, output the values of the variable `back` when the **breakpoint** is reached for the first eight times. You should submit your answer on Canvas - Quiz.

P.S. Actually, the sample code is the solution to exercise 5 :)

Example Answer:

```
1st time: back = 6
2nd time: back = ...
3rd time: back = [...]
...
8th time: back = [...]
```

Sample Code:

```
arr = [8 6 5 1 4 2 7 3];
fprintf('%d\t', mergesort(arr));
fprintf('\n');

function sorted = mergesort(arr)
    % sort vector arr into ascending order
    if length(arr) <= 1
        sorted = arr;
    else
        mid = floor(length(arr)/2);
        front = mergesort(arr(1:mid));
        back = mergesort(arr(mid + 1:end));
        % ==============
```

```
        %   breakpoint
        % ==============
        sorted = merge(front, back);
    end
end

function res = merge(a, b)
    % Given sorted vectors a and b,
    % return sort([a b])
    n = length(a) + length(b);
    res = zeros(1, n);
    a = [a inf]; b = [b inf];
    ia = 1; ib = 1;
    for i = 1:n
        if a(ia) <= b(ib)
            res(i) = a(ia);
            ia = ia + 1;
        else
            res(i) = b(ib);
            ib = ib + 1;
        end
    end
end
```

Guidelines for exercises

Generally (unless specified otherwise):

- *Strings* mean *char arrays*.
- You do not need to worry about the input range. There is no need to optimize your algorithm. Time or space complexity is not inspected.
- Use the default `double` data type. It will be enough to handle all input.
- You do not need to check for erroneous input.
- Use `disp()` for all output.
- Always remember to use semicolons (`;`) to suppress intermediate output.
- Use empty strings as input prompts.
- Do not use JOJ as a testing tool. Test your program first on your own computer before submitting to JOJ!

List of exercises

Ex#	Title	Weight
Skill	Debugging	***(8 marks)***
1	Simple recursion	***(20 marks)***
2	Tower of Hanoi	***(20 marks)***
3	Pascal's triangle	***(20 marks)***
4	*Arbitrary precision integer addition	***(15 marks)***
5	*Mergesort	***(2 marks)***
6	Demo Project 1 Milestone 2	***(15 marks)***

> ** means advanced exercises.*

Basic exercises

Ex1. Simple recursion (20 marks)

Given a positive integer $n \leq 200$, we define a set S of tuples (i.e., ordered lists) with the following rules:

1. $(n) \in S$.
2. If $l = (k_1, k_2, \ldots, k_n) \in S$, for all $k_0 \in \mathbb{Z}^+$ such that $k_0 \leq k_1/2$, $l^* = (k_0, k_1, k_2, \ldots, k_n) \in S$.

Given the number n specified by the user, output the size of the corresponding set S.

Example: If $n = 6$, we can first confirm that the ordered list $l = (6) \in S$, according to rule 1. According to rule 2, from this l, we can calculate all possible k_0, where $k_0 \in \mathbb{Z}^+$ and $k_0 \leq k_1/2 = 6/2$. So $l^* = (1, 6), (2, 6), (3, 6) \in S$. From l^* of size two, we can further induce new l^* of size three. Thus,

$$S = \{(6), (1, 6), (2, 6), (1, 2, 6), (3, 6), (1, 3, 6)\}.$$

There are 6 elements in S, so you should output 6 if the input is 6.

Hint: Use recursion. Consider the following pseudocode:

```
function result = generate_set(current_list)
    % Return the number of lists that ends with current_list.
    result = 1;
    if current_list begins with 1
        return;  % base case, recursion ends here
    end
    for all positive integers i less than or equal to current_list(1)/2
        get new_list by adding i to the front of current_list;
        add generate_set(new_list) to result;
    end
end
```

`generate_set([n])` yields the set S. *Think about why!*

We will also use this problem to demonstrate several ways of debugging.

Sample test cases

Input:

```
6
```

Output:

```
6
```

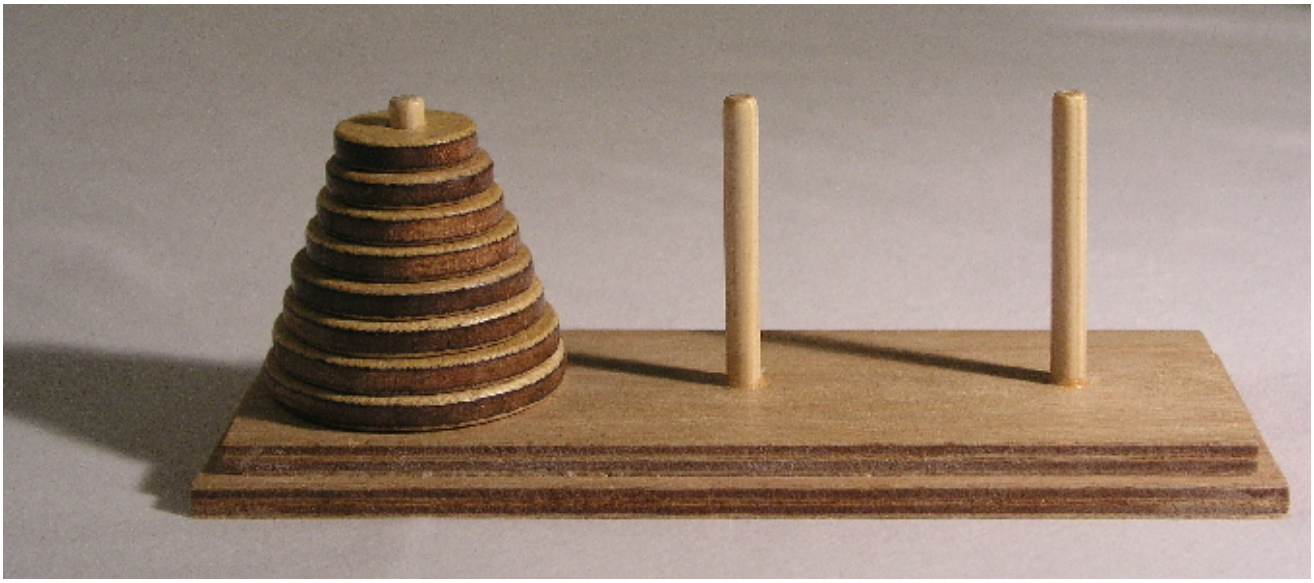
Input:

```
15
```

Output:

```
26
```

Ex2. Tower of Hanoi (20 marks)



(Image credit: Wikipedia)

Tower of Hanoi is a classical game which involves three rods and some disks with different diameters and concentric holes at the centers. The disks can be moved from one rod to another. Initially, the disks are placed on rod 1 in the order of decreasing diameter from bottom to top, as the figure shows. The goal is to move the whole stack of disks from rod 1 to rod 3 under the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper-most disk from one of the stacks and placing it on top of another stack or an empty rod.
- No disk may be placed on top of a disk that is smaller than it.

Solve a Tower of Hanoi with $n \leq 10$ disks. n is given by user input. Output each step in order in the format `s -> t`. For example, moving a disk from rod 1 to rod 2 is expressed as: `1 -> 2`.

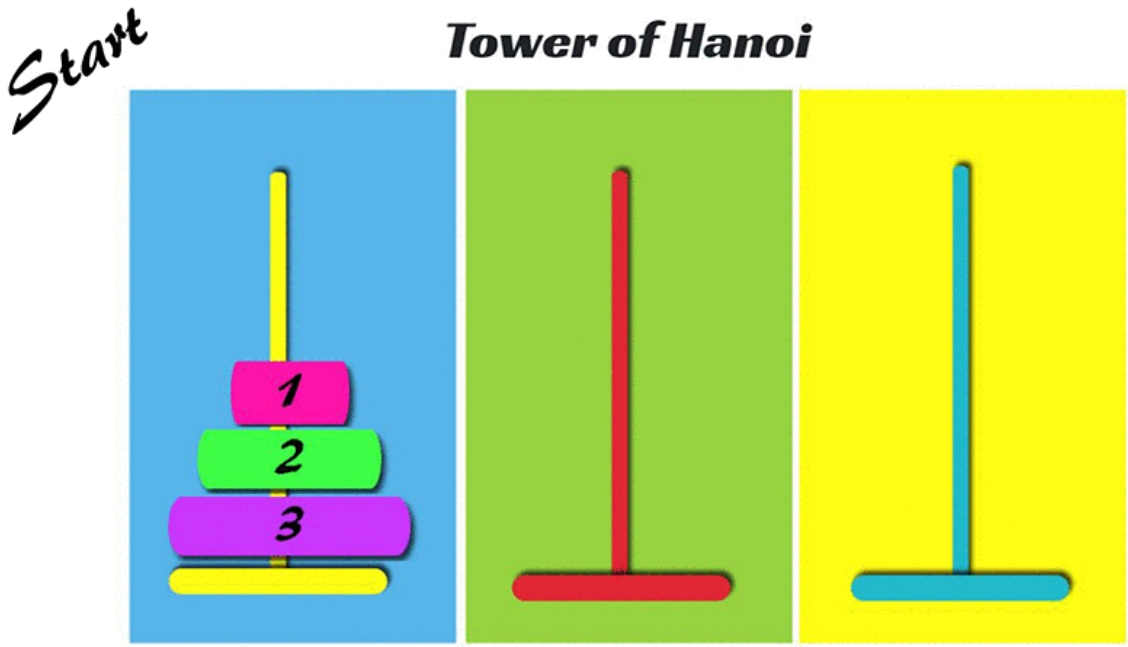
Since the number of steps needed is $2^n - 1$, large ns will lead to very long output. Be careful when testing! We recommend you to test small values only.

Hint: This problem can be solved using loops, but recursion is more suitable for it since solving a Tower of Hanoi with $n + 1$ disks can be easily reduced to solving one with n disks (recall the concept of recursion). The recursive solution is given below. You may compare it with the iterative solution on the Wikipedia entry and see why recursion is preferred.

Solution: Suppose that we know how to solve a Tower of Hanoi with $n - 1$ disks, we can solve one with n disks using the following method:

1. Move the top $n - 1$ disks from rod 1 to rod 2. (Use recursion: how to move $n - 2$ disks?)
2. Move the bottom (n^{th} and also the largest) disk from rod 1 to rod 3.
3. Move the $n - 1$ disks from rod 2 to rod 3. Since the largest disk already placed at the bottom of rod 3 is larger than all the other $n - 1$ disks, it will not interfere with this process.

The recursion stops when we reach the base case $n = 1$ where we just need to move the only disk to rod 3.



(Image Credit: Parthasarathi RV)

Code skeleton: You may begin with the function definition given below. In the implementation, since we do not always move stacks of disks from rod 1 to rod 3, it is better to call rod 1, rod 2 and rod 3 the *source rod*, the *auxillary rod*, and the *target rod* respectively.

```
function move_hanoi(n, source_rod, aux_rod, target_rod)
    % Move n disks recursively from source_rod to target_rod.
    % (source_rod, aux_rod, target_rod) is a permutation of {1, 2, 3}.
    % Base case: n = 1.
end
```

Sample test case

Input:

```
3
```

Output:

```
1 -> 3
1 -> 2
3 -> 2
1 -> 3
2 -> 1
2 -> 3
1 -> 3
```

Ex3. Pascal's triangle (20 marks)

Pascal's triangle is a triangular array of binomial coefficients $\binom{n}{m}$, sometimes also denoted by C_n^m. It can be used to easily calculate the binomial coefficients by hand when there were no computers. It has different names in different cultures. In China, it is called Yang Hui's triangle (杨辉三角) (Yang Hui discovered it hundred of years before Pascal!). It has the following properties:

1. The i-th row has i numbers.
2. The first and last elements of all rows are 1.
3. For the i-th row, its j-th element is $a_{i,j} = a_{i-1,j} + a_{i-1,j-1}$. This is due to the fact that $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$. You will see this formula in VE401 when you learn about binomial distributions.

Given a positive integer $n \leq 21$ as user input, print the first n rows of the Pascal's triangle.

Requirement: Use `fprintf()` to print the numbers. Since the largest value that can appear when $n \leq 21$ is $\binom{20}{10} = 184756$, we require you to set the *field width* to 8, leaving 2 whitespaces before 184756. See the documentation for the definition of field width. Remember to add newline characters (`\n`) between rows.

Hint: You can use an $n \times n$ matrix to store the first n rows of the Pascal's triangle. Just leave the upper-right part filled with zeros.

Sample test case

Input:

```
6
```

Output:

```
     1
     1     1
     1     2     1
     1     3     3     1
     1     4     6     4     1
     1     5    10    10     5     1
```

Advanced exercises

Ex4. Arbitrary precision integer addition (15 marks)

From the lectures, we learnt that integers are stored in computers using a fixed number of *bits*. In MATLAB, the largest unsigned integer type `uint64` takes 64 bits (or 8 bytes) to store an integer between 0 and $2^{64}-1$, which is large enough in most situations. But what if we do need to deal with larger integers?

One approach is to represent larger integers as a vector of its digits. For example, the number 1234567890 can be represented as a row vector `[1 2 3 4 5 6 7 8 9 0]`. Since a vector can be arbitrarily long (as long as there are enough memory), we can represent integers arbitrarily large.

The next question is: How do we do arithmetic operations on such integers? There are many arithmetic operations, but today we only focus on the most basic one: **addition**. Recall how you performed addition when you were in elementary school: You begin from the lowest digit, add the two numbers on this position together and *carry* to the next digit if the sum is greater than 10, and repeat this process until you reach the highest digit, with the carried 1-s also taken into account when adding the two numbers on higher digits. The digit-wise sums modulo 10 concatenated is then the sum of the two integers. This procedure is quite mechanical and you can easily implement it in MATLAB.

Given two **natural numbers** $s, t < 10^{150}$ as user input (use the parameter `'s'` to read them as strings), output their sum. You need to output a vector as a number using `fprintf()`. End your output with a newline character `'\n'`.

Hints:

- Try `'12345'-'0'`. Review the slides about ASCII encoding if you do not understand how it works.
- Use `reverse()` to reverse a char array, or `flip()` to reverse a vector
- Use 0's to fill up the higher digits of the shorter integer so that the two integers have the same length.
- Two n-digit integers may add to an $(n+1)$-digit integer.

Sample test cases

Input:

```
99999999999
99999999999
```

Output:

```
199999999998
```

Input:

```
111111111111111111111111
222222222222222222222222
```

Output:

```
333333333333333333333333
```

Ex5. Mergesort (2 marks)

Sorting is useful in many situations. Sorted data is more easy to process. There are many algorithms to sort data, among which merge sort (or spelled as mergesort) is a fairly efficient one that employs the idea of *divide and conquer*. It can be implemented using either recursion or loops, but recursion is preferred.

> *In this exercise, we use the ascending order.*

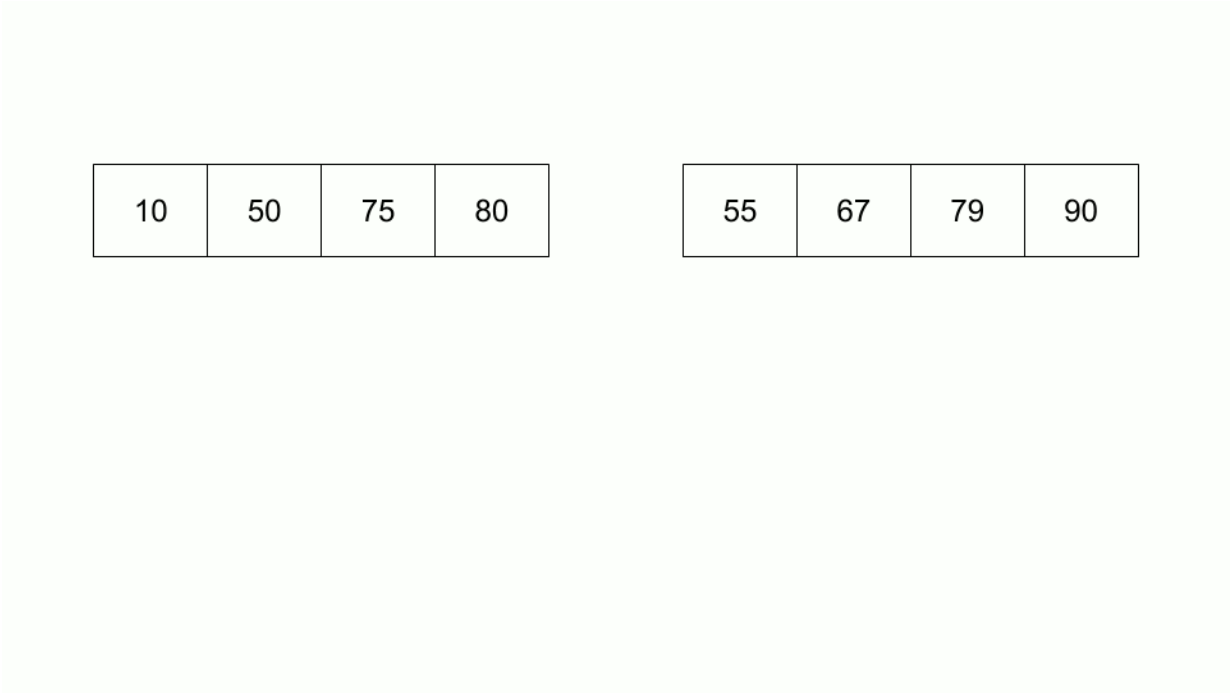
Mergesort has the word "merge" in its name because it mainly involves merging two sorted lists into a new, longer sorted list. Consider two sorted sublists `[1 4 5 6 8 10]` and `[2 4 4 7 8 9]`. How do you efficiently merge them to get the new sorted list `[1 2 4 4 4 5 6 7 8 8 9 10]`? You begin with an empty list and iteratively compare the first element of the two sublists, moving the smaller one to the new list until the two sublists are all empty.

After we know how to merge two sorted sublists, sorting a list becomes easy. We just need to first break down the list to sublists of lengths 1 (lists that contain only 1 element are considered sorted), then repeatedly merge these sublists until we are finally left with only one sorted sublist, i.e., the sorted version of the original list.

Illustration:

1. We are given a list `[8 6 5 1 4 2 7 3]`.
2. We first break it down into: `[8] [6] [5] [1] [4] [2] [7] [3]`.

3. We divide these sublists into groups of 2: (`[8] [6]`), (`[5] [1]`), (`[4] [2]`) and (`[7] [3]`).
4. For each group, we merge the two sublists. After this step, we have `[6 8] [1 5] [2 4] [3 7]`.
5. We repeat step 3 and 4 to get `[1 5 6 8] [2 3 4 7]`.
6. We merge these two sublists:
    1. We create an empty new list `[]`, referred to hereinafter as the *target list*.
    2. The first elements of the two sublists are `1` and `2` respectively. The smaller one is `1`, thus we move `1` to the target list. We now have: `[5 6 8] [2 3 4 7]` and the target list `[1]`.
    3. We repeat step 2: `2` is smaller than `5`, thus we move `2` to the target list. We now have: `[5 6 8] [3 4 7]` and the target list `[1 2]`.
    4. Repeat step 2 until we have: `[8] []` and the target list `[1 2 3 4 5 6 7]`. Since the second sublist is empty now, we can just concatenate the target list and the first sublist to get: `[1 2 3 4 5 6 7 8]`.



(Image Credit: Shahadmahmud)

7. We get the sorted list `[1 2 3 4 5 6 7 8]`.

For this exercise, take a vector of integers and use mergesort to sort it. Output the sorted vector using `fprintf()`. You may use,

```
fprintf('%d\t', mergesort(res_arr)); % \t means tab
fprintf('\n');
```

Sample Test Cases

Input:

```
[4 2 8 1 7 6 3 5]
```

Output:

```
1   2   3   4   5   6   7   8
```

Demo project 1, Milestone 2 (15 marks)

Official updates to the project description file will be posted on Piazza.

For this lab, you are supposed to complete Milestone 2. You need to:

- Write a function to update the acceleration values (i.e. Finish `updateAcc()` in `evolve.m`).
- Write a function `mutualCrash()` to deal with crashes among balls.
    - You should first finish `singleMutualCrash()` , then call this function for all possible collisions.
    - Check related formula in project description file: *elastic collision between balls*.
- Write a function `boundaryCrash()` to deal with crashes with boundaries.
    - You should complete `singleBoundaryCrash()` first.
    - If you are not sure about the condition for collision, try to draw a figure by hand.

Please download the code skeleton for this milestone from Canvas and follow the instructions in the comments.

Try to catch up with instructor because **you will need milestone 2 code for milestone 3!**

References

Lab 3 Worksheet, VG101 (21SU).