# Ground-Up Computer Science

Chapter 2

(December 20, 2021)

Yin Wang

# 2 Recursion

A: How was the exercises?

B: They are hard, but interesting!

A: Don't feel frustrated. They are just warm-ups. The following exercises may be easier.

B: They are like fun puzzles. I sweated, but not frustrated.

A: Good warm-ups. Today we will proceed towards the concept of *recursion*, which is very important to computer science and math.

B: I heard of recursion, but never really learned it.

A: Recursion is a deep topic. Few programmers learned recursion in its full strength. Having been using it for many years, I still occasionally have new discoveries of recursion.

B: That sounds profound.

A: Don't worry. We will digest it in small pieces.

B: Good.

### Boolean and string data types

A: To introduce recursion, we will first introduce a new concept called *conditional branch*, or just *branch*. But conditional branch depends on a new data type called *boolean*, so we will look at *boolean* first.

B: It seems that there is a dependency *boolean -> conditional branch -> recursion*. But first may I ask, what is a *data type*?

A: Let me explain this with examples. For example, *number* is a data type. We used numbers in the first lesson. The expressions `2`, `3`, `2 * 3`, `1 + 2 * 3`, ... Their values are all numbers.

B: Right.

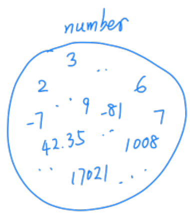
A: We call those expressions `2`, `3`, `2 * 3`, `1 + 2 * 3`, ... *number typed* expressions.

B: Because their values are numbers?

A: Right. Is `square(3)` is also a number typed expression.

B: Yes, because the value of `square(3)` is 9, which is a number.

A: Excellent. You may think of the data type *number* as a bag containing all numbers.



B: I saw this kind of picture before. It looks like a *set* in math.

A: It seems that you have learned the concept *set*. Yes, you may think of a data type as a set.

B: But I didn't learn much about sets. Actually I forgot most of it.

A: Don't worry. Think about the concept "clothing", which includes all kinds of clothes. You may think of *clothing* as a set. Similarly, when we talk about "humans", we mean the set containing you, me and all other people. *Human* is also a set.

B: A set feels like a bag of things.

A: Yes, but this "bag" is not physical. It does not physically contain things. It is something abstract. It lives in our head.

B: Now I have a better understanding of the word *abstract*.

A: To get a feel of the data type *boolean*, try this expression

```
2 > 3
```

B: I got `false` from console.

A: `2 > 3` is false, right?

B: Right. `2 > 3` is not true.

A: Try `2 < 3` then.

B: It says `true`, because `2 < 3` is truth.

A: `2 > 3` and `2 < 3` are called *comparison expressions* because they compare two values. The value of comparison expressions are either `true` or `false`.

B: I noticed that `2 > 3` and `2 < 3` look very much like `2 * 3`. There are `2` and `3`, and an operator (`>`, `<` or `*`) in the middle.

A: Good observation. Actually there is not much difference between `>`, `<` and `*`. We call `>`, `<` and `*` *binary operators* because they have two inputs. The difference is that the outputs of `*` are numbers, but the outputs of `>` and `<` are booleans. We also have `+`, `-`, `/` etc. They are all *binary operators*.

B: Are there other comparison expressions?

A: Yes. You may try these:

```
2 <= 3         // 2 is less or equal to 3
2 >= 2         // 2 is greater or equal to 2
3 == 3         // 3 is equal to 3
2 != 3         // 2 is not equal to 3
2 * 3 == 6     // 2 * 3 is equal to 6
3 == 4 - 1     // 3 is equal to 4 - 1

var y = 2 * 3;
y < 8          // y is less than 8
```

B: What are those double slashes `//` ?

A: The characters after the double slash `//` are called *comments*. JavaScript will ignore whatever you write after `//` until end of the line, so you can write additional information there to explain the code.

B: For in-class experiments, do I just type the parts that are not comments?

A: Yes. You may read the comments, but you don't need to type them into console. Now try them and let me know what are the values of those *comparison expressions*.

B: They are either `true` or `false`.

A: Here `true` and `false` are called *boolean* values. They are values of the *boolean* data type. We call `true`, `false`, `2 < 3`, `3 == 3` etc. *boolean typed* expressions.

B: Why do we write `==` for equality? Why not just `=`?

A: This is because the single equal sign `=` has already been used for variable definitions, for example `var x = 2 * 3`, so we have to use some other symbol for equality. JavaScript chose to use `==`, so did many other languages.

B: I see. `=` and `==` look quite similar.

A: Yes, so you must be very careful. Never write `=` when you mean `==`, otherwise you may cause dangerous *bugs* (computer term for mistakes).

B: What bugs can this cause?

A: It is better to think about this yourself and let me know your thoughts.

B: *(Discuss with your teacher about this.)*

A: There are no other values in the *boolean* data type, except `true` and `false`.

B: That sounds reasonable, because a comparison is either `true` or `false`.

A: Actually we can say "boolean type" for short of boolean data type. Can you draw a picture of the *boolean* type, similar to the picture of the *number* type?

B: Here it is, a bag containing just `true` and `false`.

A: Very good. Boolean is essential for conditional branches. That is why I introduce them first.

B: What is a *conditional branch*?

A: Let me show you by example. The following `abs` function can compute the *absolute value* of the input `x`. Its function body contains a *conditional branch* statement, marked in orange.

```
function abs(x)
{
  if (x < 0)
  {
    return -x;
  }
  else
  {
    return x;
  }
}
```

B: I can see that.

A: You may think of the orange part in English: "**If** (`x` is less than 0), **then** {the value of `abs(x)` is `-x`}, **otherwise** {the value of `abs(x)` is `x`}."

B: That makes more sense now. It is not very different from English.

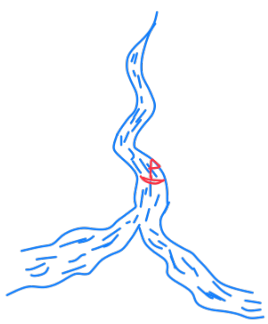
A: Constructs in programming languages are very similar to natural languages. Programming languages are just more precise in syntax.

B: All languages are similar. Some are more precise than others.

A: Right. Depending on whether `x < 0` is true or false, this conditional branch statement may execute either `return -x` or `return x`, but not both.

B: Only one branch is executed?

A: Yes. It is like branches of a river. A boat can go down only one of the branches, never both.



B: That is clear. The condition is how we decide which branch to take?

A: Correct. The general form of a conditional branch is like this:

```
if (condition)
{
```

```
  // code to be executed when condition is true
}
else
{
  // code to be executed when condition is false
}
```

Here *condition* must be a boolean typed expression.

B: Like `x < 0`, `2 >= 3`, `t == 2 * 3` etc?

A: Right. This is why I told you about the *boolean* type.

B: Now these are connected.

A: We use the value of the condition to decide which way to go, so the condition must be evaluated before we take one of the branches.

B: There seems to be an order of evaluation. Some expressions must be evaluated before others.

A: Yes. We have a similar evaluation order in the variable definition `var x = 2 * 3;`, where `2 * 3` must be evaluated before we create the variable `x`.

B: I also remember that we must first evaluate the operands of a function call.

A: Right. We will reinforce your understanding of evaluation order later, so don't worry if you can't remember them now.

B: Okay.

A: You can create functions that return boolean values too. For example, you can define a function which will return `true` if the input temperature (`temp`) is over 30 celsius.

```
function hot(temp)
{
  return temp > 30;
}
```

B: That is understandable, because `temp > 30` is similar to `temp * 2`, which is just a normal value, so we can use it as output. Can I write something like `if (hot(42)) { ... } else { ... }`?

A: Yes. The condition can be any boolean typed expression, including `hot(42)`.

B: This is like standardized machines. I put smaller components together to make bigger components.

A: Yes. This way of building things by combining smaller pieces is called *modular design*. You don't take apart the components after they are built. You just put them together.

B: This is like building cars or airplanes.

A: Exactly the same idea, very clever! Now we go on. We will write a bigger function with three branches in it. This function `sign` will return the sign of its input number. For example:

sign(-4) returns -1

sign(4) returns 1

sign(19) returns 1

sign(-28) returns -1

sign(0) returns 0

Can you see the pattern?

B: `sign` returns -1 for negative numbers, 0 for zero, and 1 for positive numbers.

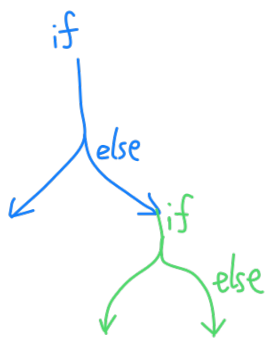
A: Yes. Now write this function using conditional branches.

B: I have some trouble here

```
function sign(x)
{
  if (x < 0)
  {
    return -1;
  }
  else
  {
    // ...
  }
}
```

The conditional branch statement can have only *two* branches, but here we have *three* cases: negative, zero and positive.

A: Here is a hint: you may have another conditional statement inside the branches of a conditional statement. For example, you may put it into the else branch.




B: Nice. A branch of a river can have branches too. This means I can write this?

```
function sign(x)
{
  if (x < 0)
  {
    return -1;
  }
  else
  {
    if (x == 0)
    {
```

```
      return 0;
    }
    else
    {
      return 1
    }
  }
}
```

A: Excellent. When you write a conditional statement inside a branch, you get three branches.

B: Yes. One of the outer branch is split into two, thus totally three branches.

A: But this way of writing three branches looks complicated. When we have many branches the code will be hard to read.

B: Are there clearer ways?

A: There is a special syntax rule which we may use here. The rule is that whenever you have an *if* immediately inside the *else* branch, you may omit the braces for the else branch.

For example, in the above code you may delete these braces (marked).

```
if (x < 0)
{
  return -1;
}
else
{
  if (x == 0)       // branch immediately inside else
  {
    return 0;
  }
  else
  {
    return 1
  }
}
```

After that, you may put the second `if` next to the `else`, so it looks like "`else if`".

```
if (x < 0)
{
  return -1;
}
else if (x == 0)
{
  return 0;
}
else
{
  return 1
}
```

B: Indeed that looks prettier, and it is clear that we have three branches.

A: But remember that this simplification requires that no other code goes between `else` and `if`, otherwise you have to write the braces.

B: Got it.

A: So we have finished learning *conditional branches*.

B: There are four basic building blocks now. *Variable*, *function*, *call* and *conditional branch*.

A: Yes. Just four of them. Now we proceed and use them to write programs that were not possible before.

B: Exciting!

### fact: a recursive function

A: Remember that our main topic today is recursion, so we proceed to write our first recursive function.

B: Okay.

A: Do you know the *factorial* function?

B: I learned factorial in math class, sort of.

A: Don't worry. I will show you by example:

The factorial of 5 is `5 * 4 * 3 * 2 * 1`.

The factorial of 4 is `4 * 3 * 2 * 1`.

The factorial of 3 is `3 * 2 * 1`.

And so on...

B: I see it now. The *factorial of n* is the product of every number from 1 to *n*. It is written as *n!* in math, so *5!* means 5 * 4 * 3 * 2 * 1.

A: Notice that *n!* is in math language. In JavaScript *n!* is written as `fact(n)`, which is a function call.

B: Can I think of math's factorial operator *!* as a function with one parameter.

A: That is exactly what it is. If `!` were a variable name in JavaScript, then we would write *n!* as `!(n)`. Unfortunately we can't use `!` as a variable name in JavaScript, so we write `fact(n)` instead.

B: I think `fact(n)` looks better than `n!` or `!(n)` because I can immediately know that it is a function call.

A: I think so too. With this preparation, now we will write a function which computes the factorial of `n`. It should be something like this:

```
function fact(n)
{
  // TODO
}
```

Don't write it yet. I will first give you some guidance.

B: Okay.

A: Written as math, we know that

5! = 5 * 4 * 3 * 2 * 1

4! =       4 * 3 * 2 * 1

Is it true that *5! = 5 * 4! ?*

B: Yes.

A: Is it true that *4! = 4 * 3! ?*

B: Yes.

A: Is it true that *n! = n * (n - 1)!* for any natural number n?

B: Yes.

A: Notice that when we say *natural numbers*, we include zero. How about zero?

B: Ah, I was wrong. For zero we can't have *n! = n * (n - 1)!* because then *(n - 1)!* would be *(-1)!*. Factorial is not meaningful for negative numbers.

A: What is the value of *0!* then?

B: Zero?

A: Actually *0!* is defined to be 1 in math.

B: Oh? How can *0!* be 1?

A: This is for simplicity. If we define *0!* as 1, then we can have *1! = 1 * 0!*, which is in the form *n! = n * (n - 1)!*. This can simplify our reasoning.

B: I don't see how it is simpler.

A: Think about it this way:

1. For zero, we have *0! = 1*.
2. For any number except zero, we have *n! = n * (n - 1)!*.

If we defined *0!* as 0, then we need one more special case for 1 here.

B: I see. If we define *0!* as 1, then we have just two cases to think about.

A: We can use a conditional statement to represent the above two cases.

B: Two branches, two cases.

A: Right. Can you translate the above math definition of factorial into JavaScript?

B: Here it is

```
function fact(n)
{
  if (n == 0)
  {
    return 1;
  }
  else
  {
    return n! = n * (n - 1)!;
  }
}
```

A: That is not right. "*We have n! = n * (n - 1)!*" doesn't mean that you just write `return n! = n * (n - 1)!`. Do you see the problem?

B: Oh, I forgot. *n!* is math's language. In JavaScript we write `fact(n)` for *n!, so (n - 1)!* should be written as `fact(n - 1)`. The last line should be

```
return fact(n) = n * fact(n - 1);
```

A: This is still not correct. Remember that after the `return`, you should write an expression that is the *output value* of the function, but `fact(n) = fact(n - 1)` is not even a valid expression in our language. It is a syntax error.

Think about this carefully. What should be `fact(n)`'s output in the recursive case?

B: `return n * fact(n - 1)`.

A: Right. Don't be misled by math's language. Now write the function in full.

B: Here it is.

```
function fact(n)
{
  if (n == 0)
  {
    return 1;
  }
  else
  {
    return n * fact(n - 1);
  }
}
```

A: Correct.

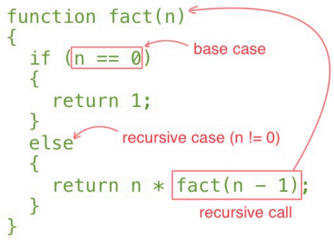
## Anatomy of a recursive function

A: Take a careful look at `fact`. How is it different from the functions in Lesson 1?

B: It calls itself inside its own function body, like a cat chasing its own tail.

```
function fact(n)
{
  if (n == 0)
  {
    return 1;
  }
  else
  {
    return n * fact(n - 1);
  }
}
```

A: Nice analogy. The concept "calling oneself" is what we call *recursion*. We call `fact` a *recursive functions*, and the call `fact(n - 1)` a *recursive call*. Take a look at this picture.




B: One picture is worth more than a thousand words.

A: The branch without recursive call is called a *base case*. Here `fact` has one base case (`n == 0`), but other functions may have more than one base cases.

B: What do you call the branches with recursive calls?

A: They are called *recursive cases*.

B: I can see a recursive case in the picture. Must we have both a *base case* and a *recursive case* in a recursive function?

A: Yes. A base case is where the function stops calling itself, so you must have at least one base case, otherwise the function will keep calling itself and go into *infinite loops*.

B: Can we demonstrate how this could happen?

A: Yes, you can. Try deleting the base case of `fact`, leaving only the recursive case. Then do substitutions repeatedly on `fact(5)`.

B: The erroneous `fact` function looks like this:

```
function fact(n)
{
  return n * fact(n - 1);
}
```

*(Write your substitution of `fact(5)` using the erroneous definition, and send it to the teacher.)*

A: Now you can clearly see that you must have a base case.

B: Yes.

### The evaluation process of fact(5)

A: Let's take a closer look at the process when we evaluate a call to fact. We use a small example `fact(3)`. Always use small examples just enough to show the ideas, because complex examples often confuse us.

B: Also they are more work to do.

A: You may use our old friend substitution for this. First do a substitution of `fact(3)`. Of course this time we use the correct definition of fact.

B: Like this?

```
if (3 == 0)
{
  return 1;
}
else
{
  return 3 * fact(3 - 1);
}
```

A: For our purpose, you may further *reduce* this. For example, here `3 == 0` must be false, so you can just go to the else branch and have `3 * fact(3 - 1)`. Because you know `3 - 1` is 2, you can reduce the whole thing to

```
3 * fact(2)
```

This simple form will aid our thinking.

B: I see. I don't need to show every tiny step. I see you used the word *reduce*. Does that mean the expression gets smaller?

A: Yes, usually smaller. You can imagine something shrinking. Now you can write out the substitutions step by step.

B:

```
fact(3)
3 * fact(2)
3 * 2 * fact(1)
3 * 2 * 1 * fact(0)
3 * 2 * 1 * 1
6
```

A: Don't go directly to 6 in the end. There are three multiplications in `3 * 2 * 1 * 1`. Which one should we compute first?

B: `3 * 2` ?

A: No. Think about it this way. Why did we get `3 * 2 * 1 * 1`? Because we wanted to compute `fact(3)`, which is substituted to `3 * fact(2)`. But until we have the value of `fact(2)`, we can't compute the multiplication `3 * fact(2)`.

B: This means we must compute `3 * fact(2)` last?

A: Actually, if you put parentheses on multiplications in your substitutions, you will see the order clearly.

B: I just put every multiplication into parentheses. Indeed this is a lot more clear.

```
fact(3)
(3 * fact(2))
```

```
(3 * (2 * fact(1)))
(3 * (2 * (1 * fact(0))))
(3 * (2 * (1 * 1)))
```

A: Right. Is that interesting?

B: Very interesting. I didn't know that parentheses can help so much. It is now all clear without much effort.

A: Remember this trick. Sometimes writing out every parentheses explicitly can help. Take a look at the evaluation process. Do you see any problems?

B: As I can see, the intermediate expressions can get very long.

A: Yes. Consider computing `fact(1000)`. The intermediate steps will grow to 1000 in length, `1000 * 999 * 998 * ... * 3 * 2 * 1 * 1`.

B: Does it take space to store this expression?

A: Yes. Nothing comes for free. Every number has to be stored somewhere, otherwise we would not know what to multiply.

B: That seems to be a big problem.

A: This way of writing `fact` function is not *efficient* regarding to storage space, but our purpose is just to show the anatomy of recursive functions, and `fact` serves as a very good first example. In practical programs, you don't want to write it this way.

B: I will remember `fact`, our first etude of recursive functions.

**Another recursive function (fib)**

A: Now we take a look at another recursive function `fib`, where `fib(n)` will compute the nth fibonacci number.

B: Fibonacci number. I heard of it too, but...

A: Let me remind you. The first fibonacci number is 0, the second fibonacci number is 1, then every next fibonacci number is the sum of the previous two fibonacci numbers.

B: Can you give me an example?

A: Here is the fibonacci sequence:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

B: I see, in math language,

fib(0) is 0

fib(1) is 1

fib(2) = fib(0) + fib(1), which is 1

fib(3) = fib(1) + fib(2), which is 2

fib(4) = fib(2) + fib(3), which is 3

fib(5) is fib(3) + fib(4), which is 5

fib(6) is fib(4) + fib(5), which is 8

fib(7) is fib(5) + fib(6), which is 13

fib(8) is fib(6) + fib(7), which is 21

... ...

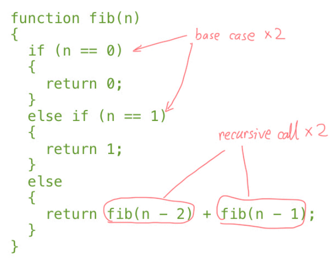
A: Yes. Using experience of `fact`, can you write a recursive function `fib`, which computes the nth fibonacci number?

B: Here it is.

```
function fib(n)
{
  if (n == 0)
  {
    return 0;
  }
  else if (n == 1)
  {
    return 1;
  }
  else
  {
    return fib(n - 2) + fib(n - 1);
  }
}
```

A: Perfect. Can you tell me how many *base cases* and *recursive calls* are there?

B: Two bases. Two recursive calls.




A: Nice picture. Why do we have two base cases?

B: I'm not sure. We are given the first two numbers. They are not computed but just given, so I used them as base cases.

A: Think about this. We need to compute the other numbers from the previous *two* numbers. The recursion will end up needing the values of `fib(0)` and `fib(1)`, so you can't have just one base case.

B: Is there a systematic way in which I can know how many base cases to write?

A: The trick is to look at the recursive calls. See how the recursive call's input parameters are different from the function's parameters, and then figure out the values in which they end up.

B: I don't understand this.

A: It is a bit early to explain this in full. We will come up with a systematic way of thinking about recursion in the next class after we have more ways to write recursive function. For exercises of this class, you may just mimic `fact` and `fib`.

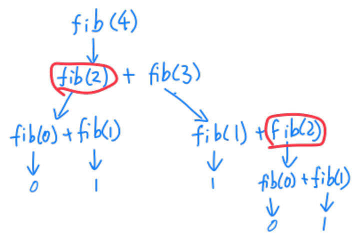
B: Okay.

A: Can you see that our way of writing the `fib` function has a big problem?

B: Does it also take a lot of storage space, like `fact`?

A: Not exactly the same problem. You can use `fib(4)` as an example, try drawing a graph showing the substitutions of the recursive calls. Draw an arrow connecting *each* recursive call to its substitution.

B: I can draw it like this. Because there are two recursive calls, the graph seems to be "branching" into a tree.

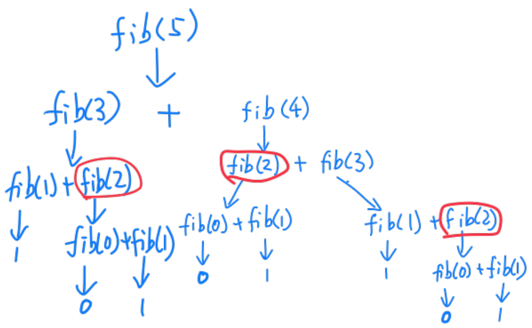



A: Good observation. Can you see how many times `fib(2)` is evaluated?

B: Twice. I have put two circles on them.

A: Right. Try expanding this graph into a graph for `fib(5)`, and again see how many times `fib(2)` is evaluated.

B: Three times. The number of times we evaluate `fib(2)` seems to be growing with bigger `fib(n)`.




A: Can you see the problem?

B: Yes. We are wasting time by computing some expressions repeatedly.

A: Have you noticed other such repeated computations?

B: Yes. `fib(3)`, `fib(1)`, `fib(0)`, ...

A: Now that we see the examples of `fib(4)` and `fib(5)`, try to figure out a general formula in which the number of repeated evaluation of `fib(2)` grows with `n`. For example, is it $2n$, $n^2$, or $2^n$? If you have no clue, try extending our previous examples. Draw a graph of `fib(6)`, `fib(7)`, and so on.

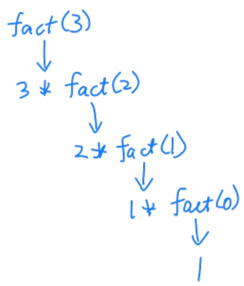
B: *(Write your answer and send it to the teacher.)*

A: So `fib` is a very slow way of implementing fibonacci numbers, but it demonstrates this type of recursion very well.

B: What type of recursion? How is it different from `fact`?

A: Try redraw a picture for `fact(3)`, in the style you just did for `fib(4)`.

B: Here it is. I can see that the graph for `fact(3)` has no branching. It is more like a chain, not a tree.




A: Yes. The recursion of `fact` is called *linear recursion*, while the recursion of `fib` is called *tree recursion*, because it looks like a tree with branches.

B: I can see a downward growing tree in `fib`'s recursion process.

A: Why is `fib` a tree recursion?

B: The branches come from `fib(n - 2) + fib(n - 1)`, so the reason of tree recursion seems to be in the number of recursive calls.

A: Good observation. Whenever you see two recursive calls in the same recursive case, that means a tree recursion.

B: That is useful observation, because tree recursions may be costly.

A: Tree recursion may not always be costly, and it is often the only way to write certain programs.

B: Some teachers told me that recursion is slow and should be avoided or converted to other ways, for example *loops*.

A: I don't want to teach "what is a loop" for now, but it is a usual misunderstanding of recursion. Recursion is usually not any slower than loops. If you can't write a fast program with recursion, then you can't write a fast program with loops or any other way either.

But recursion is usually a lot simpler than other ways, so it is not something you should avoid. You just have to master the idea. It is fundamental to computer science, math and so many natural phenomenon in the universe.

B: What a terrible misunderstanding. I can see how much those people have missed. I will do my best to master recursion.

A: Good. We have more recursive functions in later classes. Actually almost every function we will write from now on will be recursive functions!

B: Wow!

A: Okay. That's all for today's class. Here are the exercises.

B: Thank you. Have a good night!