# Function evaluation, recursion and complexity

Jia Xiaodong

Last revised August 22, 2021

## Substitution Model

- What is the substitution model?
  - Reasoning about programs
- What is the idea behind it?
  - Certain expressions are *irreducible*.
  - Computation continues until we cannot proceed further, i.e. we get something that is irreducible.
  - By performing repeated reductions, we can simplify and find the result of any given statement.

The substitution model is simple. Previously we have seen how programs are made of statements and expressions. Each of these evaluate to some result. For example `3 + 2` evaluates to `5`, whereas `'a'` evaluates to `'a'`. From here we can see that there are *irreducible* expressions.

Therefore to evaluate anything all we do is to apply some reduction rules till the expression becomes irreducible. This is fairly ordinary for arithmetic operations. However, how are statements with function calls going to be evaluated?

## Applicative Order Reduction

**What does this do?**

```
12345 % math_pow(10, math_floor(math_log10(12345)));
```

Let's try:

- `12345 % math_pow(10, math_floor(math_log10(12345)));`
- `12345 % math_pow(10, math_floor(4.09...));`
- `12345 % math_pow(10, 4);`
- `12345 % 10000;`
- `2345;`

Applicative order reduction basically like the arithmetic case — evaluate from the left and deepest expression. Here are how the steps unfold. Before you evaluate a function, all of its arguments must be evaluated first.

1. The leftmost number is already irreducible so we move on.
2. `10` is done. So the `math_floor` has to be evaluated.
3. Then `math_log` has to be evaluated first. The number it takes in is already irreducible so we can go ahead ahd do that.
4. We get `4.09` something which we then plug into `floor`, and so on. . .

To answer the question as to what this program does, it gets rid of the leading digit of any number. A more useful thing might be the stuff after the %, which gives a power of 10 with as many digits as the number you feed in. In this case we feed in `12345` with 5 digits, it will produce `10000` with 5 digits.

# Normal Order Reduction

## What does this do?

```
function sq(x) { return x * x };
function dist(x, y) { return math_sqrt(sq(x) + sq(y)) };
dist(1 + 5, 2 * 10);
```

Let's try:

- `dist(1 + 5, 2 * 10);`
- `math_sqrt(sq(1 + 5) + sq(2 * 10))`
- `math_sqrt((1 + 5) * (1 + 5) + (2 * 10) * (2 * 10))`
- `math_sqrt((6) * (1 + 5) + (2 * 10) * (2 * 10))`
- `math_sqrt((6) * (6) + (2 * 10) * (2 * 10))`
- etc.

Evaluate from left to right. Fairly straight-forward. Do note that the interpreter does not use this evaluation strategy, so this is a FYI only.

## Exercise

### Ex. 1.5

```
function p() {
    return p();
}

function test(x, y) {
    return x === 0 ? 0 : y;
}

test(0, p());
```

What does this evaluate to?

Sometimes it feels very natural to just plug values in and forget about the rules. One might forget here that `p()` is not irreducible and needs to be evaluated. Once we remember this fact, we see that this program doesn't even evaluate to anything since it never stops.

Then again, using normal order reduction you will get a value, 0. The fact that the interpreter does not give this value and instead chooses to crash serves to highlight again that we are using applicative order reduction in this course!

# Recursion

**From Wikipedia:**

Recursion (adjective: recursive) occurs when a thing is defined in terms of itself or of its type.

This gives rise to a way of solving certain problems. Certain problems exhibit the property of *optimal substructure*. This means that the method to solve a large problem is by breaking it up and solving the smaller sub-problems. Then you piece it back together.

Recursion on its own is quite simple. It is mathematical in origin. In fact the earliest programming languages did not always support recursion. Nowadays all mainstream ones do.

In any case, the main scenario where we use recursion is because you might not know how to solve the entire problem but you do have a way of solving a small problem and you know how to combine solutions.

## Recursion

- Things we need:
  - The base case, or the trivial case.
  - A relationship between the large problem and the smaller sub problems.

**Ex: Listing out $\mathbb{N}$**

$$s_0 = 0 \quad s_n = s_{n-1} + 1$$

**Ex: Fib($n$)**

$$F_1 = 1, F_2 = 1 \quad F_n = F_{n-1} + F_{n-2}$$

Let us explore more. Again, what we need is

- a small problem we know how to solve, the base case,
- and a way to combine small solutions or build up on them.

A few easy examples.

- A child is learning to count to 100. He does not know what is 100 since it's too big. It is easy to know what's 0, though. So we start with 0, and now we increment it and one day we will hit 100.
- Fibonacci numbers. If you are asked for the 100-th one, you would have to use the relationship and try to find the 99th and 98th one, and so on.

In fact here we see two different ways of using recursion. The first example is a bottom-up approach, and the second example is a top-down approach.

## Time and Space Complexities

- Why do we care?
  - We need an abstract way to talk about resources consumed.
  - We do not want to care about worldly problems like programming languages, computer architecture, CPU speed, etc.
  - We want to know how input affects it.

Since now we are on the topic of algorithms, we want to be able to tell if one is better than another. The two main resources on any computing device is time and space. However we do need a better measurement than seconds and bytes since these are going to change based on your system.

We will also definitely be taking in input in our calculations. It is also easy to imagine that larger inputs most likely use up more time. So naturally we also want to relate resources consume to input size.

## Time Complexity

- Some abstract measure of time taken for the program to run.
- How do we characterize it?
  - Number of operations performed.
  - Number of "simple" operations performed for some input size.
  - Simple operations:
    - All arithmetic e.g. `4 * 5`
    - Memory read and write e.g. `const a = 4;`
    - Conditionals e.g `if (a === 4)`
- An asymptotic bound on the number of primitive operations by *nice* functions [1].

[1] You can forget about this entirely, you will never meet a bad function in this course. This is however usually enforced because there exist pathological functions that really mess up complexity classes.

We want to know how long an algorithm is going to take — what costs time in an algorithm? The number of steps it runs. Of course you can just call the entire program a single step but that's not very useful. So there are certain steps that are defined to take 1 unit of time to complete.

Furthermore, we will see that we don't really care if the simple steps actually take 1 or 2 or 100 units of time, as long as they are guaranteed to always take the same amount of time ("constant time"). Hence there can be some lax when counting these operations.

The number of steps taken are going to vary based on the size of the input. So we want to provide some indication of the trend that the number of steps taken as we increase the size of the input.

An aside: what constitutes a single step depends on the mathematical model of your machine. For example usually it is customary to let all arithmetic operations take 1 unit time, but multiplication is always slower than addition, and on certain processors you don't have multiplication at all. There are also models of machines that can only add and subtract by 1, so it really depends.

## Space Complexity

- Some abstract measure of space taken for the program to run.
- How do we characterize it?
  - Number of symbols created.
  - Maximum number of “simple” symbols created.
- An asymptotic bound on the space required relative to input size.

The same goes for space requirements. Now we count the number of symbols created. So what is a symbol? For our purposes, it does not harm just counting anything that takes up memory: numbers, characters, functions, etc. There is some subtlety here regarding the length of the symbols, but you do not have to worry about it now.

Of course the amount of space the program uses will change while it's running. What we are looking for thus is the maximum, since that's the resources you are going to need.

Most of the time these things are very natural and common-sense to count, so there is no need to worry too much.

# Big O notation

To accomplish these things we use the Big O asymptotic notation.

| Name | Definition | Meaning |
|---|---|---|
| $f(n) = O(g(n))$ | $\exists k > 0, \exists N, \forall n > N, f(n) \leq k \cdot g(n)$ | $f$ is bounded above by $g$. |
| $f(n) = \Omega(g(n))$ | $\exists k > 0, \exists N, \forall n > N, f(n) \geq k \cdot g(n)$ | $f$ is bounded below by $g$. |
| $f(n) = \Theta(g(n))$ | $\exists k_1, k_2 > 0, \exists N, \forall n > N,$<br>$k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$ | $f$ is bounded by $g$. |

We can find some constant factor(s) such that regardless of how large the input gets (asymptotic) we can provide a bound on the function.

These are the formal definitions. There also exist a small o notation that is stricter, and you can look it up for enrichment.

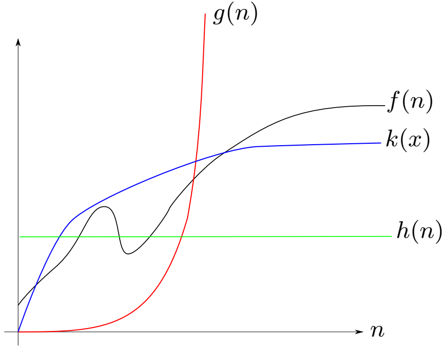
In any case you only need to remember the meaning of the notation. Here is what they are used for

- $O$: Provides a worst-case upper bound (may not be tight).
- $\Theta$: Provides a tight bound on growth rate.
- $\Omega$: Provides a best-case lower bound. This is used less for complexity analysis of an algorithm, but for denoting that some problem must take *at least* some amount of resources.

Also most times when people use $O$, frequently they actually mean $\Theta$ but it has become quite customary to just treat $O$ as giving a tight bound as well. This is because you can always give some ridiculous bound like $O(\exp(\exp(\exp(x))))$ and be always correct, but it wouldn't do you much good.

# Big O notation

## Graphical illustration



Here $f$ is our function we wish to provide a bound for.

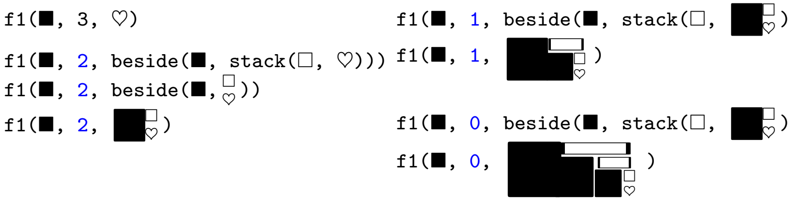
- $g$ clearly forms an upper bound on $f$ eventually. Hence we can write $f(n) = O(g(n))$.
- $h$ also forms an lower bound on $f$ eventually. Hence we can write $f(n) = \Omega(h(n))$.
- $k$ seems to provide a lower bound as well. However, notice that if we stretch $k$ vertically, it will provide an upper bound on $f$ as well. So it gives a tight bound on the behaviour of $f$, and we can write $f(n) = \Theta(k(n))$.

An aside: everyone uses $f = O(g)$ when more accurately it is $f \in O(g)$. I believe it particularly makes sense in mathematics where for example if you were performing an approximation you can write $\sin(x) = x + O(x^3)$ and using $\in$ would be odd. Stick to using $=$.

# S3 Q1

```
function f1(rune_1, n, rune_2) {
    return n === 0
        ? rune_2
        : f1(rune_1, n - 1, beside(rune_1,stack(□, rune_2)));
}
```

Evaluate `f1(■, 3, ♡)` using the substitution model.



```
f1(■, 3, ♡)

f1(■, 2, beside(■, stack(□, ♡)))
f1(■, 2, beside(■, □♡))

f1(■, 2, )

f1(■, 1, beside(■, stack(□, )

f1(■, 1, )

f1(■, 0, beside(■, stack(□, )

f1(■, 0, )
```

We use ■ to represent `square` and □ to represent `blank` and ♡ to represent `heart`. The symbols and spacing drawn here are not to scale.

This is fairly straight-forward. The only hiccup is to keep in mind what is being evaluated and what is being returned. For example you may pre-maturely finish evaluation at $n = 1$, by thinking execution ends at the next step where $n = 0$. Execution cannot end there because $n \neq 0$! This is more likely than not caused by writing in shorthand as we do here, so be careful!

## S3 Q2

```
function f2(rune, n) {
    return n === 0
        ? rune
        : stack(beside(□, f2(rune, n - 1)), ■);
}
```

Evaluate f2(♡, 3) using the substitution model.

f2(♡, 0);

♡

f2(♡, 1);

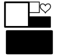
stack(beside(□, f2(♡, 0)), ■)

stack(beside(□, ♡), ■)

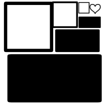
stack(□♡, ■)

f2(♡, 2);

stack(beside(□, f2( , 1)), ■)



f2(♡, 3);

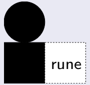


For this question you might get into quite a tangle going in head first.

Unofficially, an easier way could be to take a bottom-up approach instead. However this is no the substitution method. The proper method is left as an exercise.

# Q1

Write a function `moony_1(rune)` that outputs this:

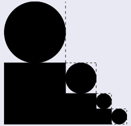


```
function moony_1(rune) {
    return stack(beside(circle, blank),
        beside(square, rune));
}
```

There are two simple ways of doing this. Shown here is one way.

## Q2

Write a function `moony_2` to recursively insert circles into the right place. Example output of `moony_2(4)`:



```
function moony_2(n) {
    return n === 1
        ? circle
        : moony_1(moony_2(n - 1));
}
```

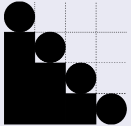
You can go about this both ways, top down and bottom up.

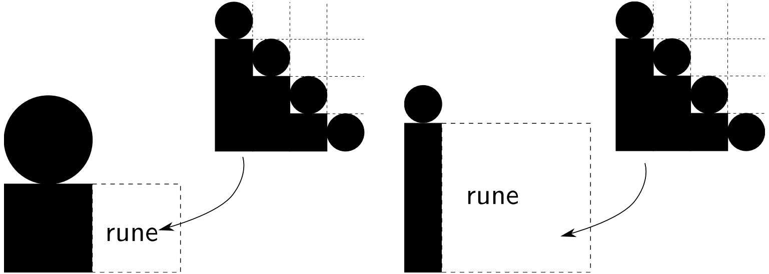
The top down way requires you to see that by squeezing the larger rune into the position of `rune` in `moody_1` you can create the image. The base case is then a circle.

The bottom up way requires you to see that the base case is just a circle. From there you can build up on the circle by using it as `rune` in `moody_1`.

## Q3

Now make the circles have the same diameters:





We do not need to modify anything else. All that matters here is some scaling. The previous questions might also not be very helpful here.

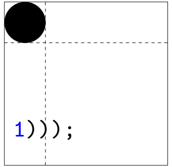
The base case occurs again at $n = 1$. Let us think now: if we have an image that has been stacked nicely, and if we were to just simply squeeze it into the bottom right corner, the ratios would be messed up. What we need here is to make the circle smaller. The extra space would be occupied by the square (now no longer a square).

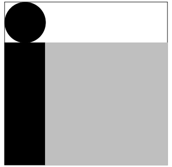
How much more space? Since every column takes up the same amount of space, then the circle has to occupy $\frac{1}{n}$ units of space horizontally and vertically. For the square below it, it will occupy $\frac{1}{n}$ units horizontally, and $\frac{n-1}{n}$ units vertically.

# Q3

## Cont.

```
function moony(n) {
    return n === 1
        ? circle
        : stack_frac(1 / n,
            beside_frac(1/n, circle, blank),
            beside_frac(1/n, square, moony(n - 1)));
}
```





# Q4

Do your functions give rise to recursive or iterative processes? What is the time and space complexities of your `moony`?

| Name | Process | Space | Time |
|---|---|---|---|
| `moony_1` | — | $\Theta(1)$ | $\Theta(1)$ |
| `moony_2` | Recursive | $\Theta(n)$ | $\Theta(n)$ |
| `moony` | Recursive | $\Theta(n)$ | $\Theta(n)$ |

Assuming all runes take up constant space, and the rune operations we used (stack and beside) also operate in constant space and time. This is quite a realistic assumption.

## Q1

```
function expt(b, n) {
    return n === 0
        ? 1
        : b * expt(b, n - 1);
}
```

- `5 * expt(b, 4)`
- `5 * 5 * expt(b, 3)`
- `5 * 5 * 5 * expt(b, 2)`
- `5 * 5 * 5 * 5 * expt(b, 1)`
- . . .

- Time: $\Theta(e)$
- Space: $\Theta(e)$

This is just exponentiation by repeated multiplication. It is similar to the factorial function we have seen previous. The analysis is similar. There are deferred operations, and we clearly see that it extends as many times as we exponentiate.

## Q2

```
function fast_power(b, e) {
    return e === 0
        ? 1
        : is_even(e)
          ? fast_power(b * b, e / 2)
          : b * fast_power(b, e - 1);
}
```

- `fast_power(3, 4)`
- `fast_power(9, 2)`
- `fast_power(81, 1)`
- `81 * fast_power(81, 0)` ✓

- Time: $\Theta(\log e)$
- Space: $\Theta(1)$

This is exponentiation by squaring.

We can answer the first question: there will be deferred operations unless $e$ is a power of 2.

As for the runtime, first we can see that for any exponent $e$ that is a power of 2, it will take $O(\log(e))$ steps. The example run shows how this can be faster than the naive exponentiation implementation. How much faster?

## Q9

Cont.

```
    return e === 0
        ? 1
        : is_even(e)
            ? fast_power(b * b, e / 2)
            : b * fast_power(b, e - 1);
```

- fast_power(3, 10)
- fast_power(9, 5)
- 9 * fast_power(9, 4)
- 9 * fast_power(81, 2)
- 9 * fast_power(6561, 1)
- 9 * 6561 * fast_power(6561, 0) ✓

For any other $e$, we can write

$$e = \sum_{j=0}^{k} a_i 2^k$$

where $k$ is the smallest integer such that $2^{k+1} > e$, and $a_i \in \{0, 1\}$. For example, $10 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$ (the more astute will recognize this as binary). The number of non-zero coefficients give us one more than the number of multiplications. $k$ gives us the number of squares.

Is this magic? If we look at $\frac{10}{2}$, we get remainder 0, which means $10 = 5 \cdot 2^1 + 0 \cdot 2^0$. Next, $\frac{5}{2}$ gives remainder 1, which means that $10 = (2 \cdot 2^1 + 1 \cdot 2^0) \cdot 2^1 + 0 \cdot 2^0 = 2 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$. Doing the same for 2, we get the expansion as shown above. Notice that this is exactly the process `fast_power` goes through! Every time there is some remainder, `fast_power` does one multiplication, and in our expansion we encounter a coefficient of 1. At every division it has to do a square, so there are $k$ number of them.

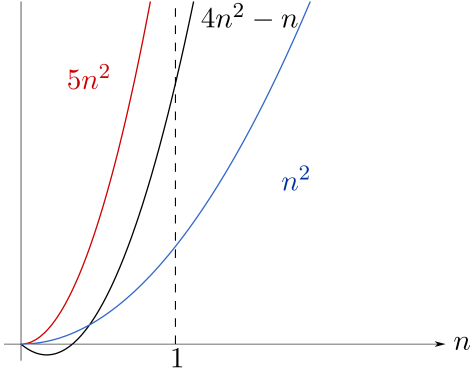
What is $k$? $k = \Theta(\log e)$.

# Q1

Show that $4n^2 - n = \Theta(n^2)$

$n_0 = 1, k_2 = 5, k_1 = 1.$

In fact one can show that for any polynomial $p_m(n)$ of degree $m$, $p_m(n) = \Theta(n^m)$.



From the graph it is easy to check that the provided witnesses support the claim. This is fine since here since the derivatives also tell us that their order will be preserved.

Let $p_m(n)$ be a polynomial of order $m$ with $n$ as the variable. To show that $p_m(n) = O(n^m)$, choose $n_0 \geq 1$ such that $p_m(n) \geq 0$ for all $n \geq n_0$. Suppose $p_m(n)$ is of the form $a_0 + a_1 n + \cdots + a_m n^m$ where $a_m$ is positive (why must it be positive?). Then let $k = |a_0| + \cdots + |a_m|$.

To show that $p_m(n) = \Omega(n^m)$, choose $n_0$ such that $a_0 + \cdots + a_{m-1}n^{m-1} + \frac{a_m}{2}n^m \geq 0$ for all $n \geq n_0$. Then $p_m(n) \geq \frac{a_m}{2}n^m$, so we take $k = \frac{a_m}{2}$.

Also, a tight bound means both directions: $f(n) = \Theta(g(n)) \iff f(n) = \Omega(g(n)) \wedge f(n) = O(g(n))$.

# Q2

Show that $\log_5(n) = \Theta(\ln n)$

$$\log_a n = \frac{\log_b n}{\log_b a}$$

Any logarithm regardless of its base is only scaled by a constant factor with respect to any other logarithm. Hence there is no need to specify a base in our big O notations involving logarithms.

Another question: what about exponentials? i.e. for example between $2^n$ and $3^n$?

## Q3

$$10n \log n \overset{?}{=} O(n^2)$$

$$10n \log n \overset{?}{\leq} 2n^2$$
$$\log n \leq n$$

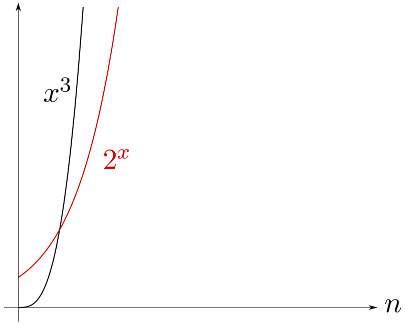
We can work backwards from the claim to check. This is fairly easy. For the proof, just reverse the workings.

The proof that $\log n$ grows slower than $n$ (of course only if we pick a suitable $n_0$) is basic calculus and is left as an exercise.

# Q4

$$n^3 \overset{?}{=} O(2^n)$$

$$n^3 \overset{?}{\leq} k \cdot 2^n$$
$$\log n^3 \overset{?}{\leq} \log k + \log 2^n$$
$$3 \log n \leq \log k + n \log 2$$



First of all keep in mind that for some number $c$, $n^c$ and $c^n$ are very different things. Also this is a question where "proof by graph" might fail.

To understand why, suppose we just pick $k = 1$, to draw the test graph as shown in the figure. Then looking at our inequality let us find the $n_0$ such that for all $n \geq n_0$ it holds true:

$$3 \log n \leq n \log 2$$
$$10 \log n \leq n$$

where $\frac{3}{\log 2} \approx 10$. Usually such equations do not have easy solutions, but today we are in luck: $n_0 = 10$. Well that doesn't seem too unreasonable, but then notice that this means $2^n$ will start to overtake $n^3$ only after $2^{10} = 1024$! That is way outside the plotting window, and you would not have noticed it unless you put in the effort to scroll that far. Hence, graphical methods are not very acceptable as proofs.

## Q5

a) $5n^2 + n = \Theta(?)$

b) $\sqrt{n} + n = \Theta(?)$

c) $3^n n^2 = \Theta(?)$

a) $\Theta(n^2)$

b) $\Theta(n)$

c) $\Theta(3^n n^2)$

For part (a) we have already shown this previously.

For part (b) note that $\sqrt{n}$ grows slower than $n$ so $\sqrt{n}+n$ grows slower than $n + n$.

For part (c), no constant grows faster than $n^2$, so it cannot be $\Theta(3^n)$. From the previous question it definitely cannot be $\Theta(n^2)$ either. So nothing can be done here.

For the next few questions, give the space and time complexities of the functions presented.

## Q6

```
function factorial(n) {
    return n === 1
        ? 1
        : n * factorial(n - 1);
}
```

- `5 * factorial(4)`
- `5 * 4 * factorial(3)`
- `5 * 4 * 3 * factorial(2)`
- `5 * 4 * 3 * 2 * factorial(1)`
- `5 * 4 * 3 * 2 * 1`

- Time: $\Theta(n)$
- Space: $\Theta(n)$

This will be easier if we list out the process that this function gives rise to. Question: why don't we evaluate the `5 * 4` term?

We can see that a call to this function results in $n$ operations being created. Then the time and space complexities are easy to determine.

## Q7

Write `factorial` that gives rise to an iterative process.

```
function _(n, res) {
    return n === 1
        ? res
        : _(n - 1, n * res);
}

function factorial(n) {
    return _(n, 1);
}
```

The deferred operations do not get evaluated until the end because they do not "touch" each other — they belong to different results. Hence the standard recipe to eliminate these is to let them "touch", by using an accumulator to store your result.

Here actually our secret little function _ does the work, and as a layer of abstraction we wrap `factorial` around it. `factorial` knows how to operate _ properly.

In any case, now there are clearly no deferred operations. The space consumption will then be $O(1)$. For time consumption, since _ runs $n$ times and its body is evaluated in constant time, `factorial` takes $O(n)$ time.

Here I use _ as a name just for fun. If you read the language specification you might find other kind of weird names you can come up with, such as `$`. In reality, these names are frequently used in other applications. In this case, since we have not covered higher order functions, one way to hide our user-unfriendly factorial function is to give it an odd name. Many languages use names starting with underscores to denote that this might be "private property".