Cheatography

Elixir Cheat Sheet

by zokratez via cheatography.com/211420/cs/45803/

Enum types

I need...	Function	Example	BigO
Execute without return	Enum.each/2	Enum.each([1,2,3], &IO.puts/1)	O(n)
Transform each element	Enum.map/2	Enum.map([1,2,3], &(&1 * 2)) # [2,4,6]	O(n)
Filter elements	Enum.filter/2	Enum.filter([1,2,3], &(&1 > 1)) # [2,3]	O(n)
Accumulate a value	Enum.reduce/2	Enum.reduce([1, 2, 3, 4], fn x, acc -> x * acc end)	O(n)
Find a value	Enum.find/2	Enum.find([1,2,3], &(&1 > 1)) # 2	O(n) (worst case)
Check if all match condition	Enum.all?/2	Enum.all?([2,4,6], &rem(&1, 2) == 0) # true	O(n) (worst case)
Check if exists inside	Enum.member?/2	Enum.member?([1,2,3], 2) # true	O(n)

Data Structure Types

Data Structure	Access Compex	Observations
list	O(n)	sorted, duplicated keys
tupple	O(1)	
map	O(1) map > 32 / O(n) map < 32	unsorted, no key duplication
keyword list	O(n)	Inserting with prepend is O(1)

Maps < 32 are sorted lists
Maps > 32 are hash-tree (HAMT)
Lists, when insert prepend for O(1), append is O(n)

Sort Algoritms

Name	Time Complex	Comments
Merge Sort	☐ O(n log n) / O(n log n) / O(n log n)	Recommended in Elixir (recursive and stable)
Insertion Sort	✓ O(n) / ☐ O(n²) / O(n²)	Good for small or semi-ordered lists
Radix Sort	O(nk) / O(nk) / O(nk)	Very fast on small integers
Bubble Sort	✓ O(n) / ☐ O(n²) / ✗ O(n²)	Inefficient, useful only for small or already sorted lists
Selection Sort	☐ O(n²) / O(n²) / O(n²)	Simple, but slow
QuickSort	✓ O(n log n) / O(n log n) / ✗ O(n²)	Better than MergeSort on average, but can degrade to O(n²)
Heap Sort	O(n log n) / O(n log n) / O(n log n)	
Counting Sort	O(n + k) / O(n + k) / O(n + k)	Only works with small numbers and limited range
Bucket Sort	O(n + k) / O(n + k) / O(n²)	Useful if data is evenly distributed

Avoid:
QuikSort because we can get O(n²) if the list is already sorted.
Insertion (O(n²)) → If the list is inverted.

Merge Sort

Merge Sort (cont)

```
> defmodule MergeSort do
  def sort([]), do: []
  def sort([x]), do: [x]
  def sort(list) do
    midpoint = div(length(list), 2)
    {left, right} = Enum.split(list, midpoint)

    new_left = sort(left)
    new_right = sort(right)

    merge(new_left, new_right)
  end
  def merge(left, []), do: left
  def merge([], right), do: right
  def merge([l | left], [r | right]) when l <= r do
    [l | merge(left, [r | right])]
  end
  def merge([l | left], [r | right]) do
    [r | merge([l | left], right)]
  end
end
MergeSort.sort([2,4,6,7,1,2,3,5])
```

Insertion Sort

```
defmodule Sort do
   def insert([], x), do: [x]
   def insert([h | t], x) when x <= h, do: [x, h | t] # Insert before if is lower

   def insert([h | t], x), do: [h | insert(t, x)] # Continue searching if is greater
   def insert ion _so rt([]), do: []

   def insert ion _so rt([h | t]) do
        sor ted _tail = insert ion _so rt(t)
        ins ert (so rte d_tail, h)
   end
end
Sort.i nse rti on_ sor t([3, 1, 4, 2])
```

Detect Anagram

```
defmodule Anagram do
   def anagra m?( word1, word2) do
        Enu m.s ort (St rin g.g rap hem es( word1)) == Enum.s ort (St rin g.g rap hem es( word2))
   end
```

Detect Anagram (cont)

```
> end
IO.inspect(Anagram.anagram?("listen", "silent")) # true
```

O(n log n) because Enum.sort

Palindrome

```
defmodule Palindrome do
   def is_pol ind rom e?( word) do
    wor d_t o_c harlist = to_cha rli st( word)
    (wo rd_ to_ cha rlist == Enum.r eve rse (wo rd_ to_ cha rlist))
   end
end
Palind rom e.i s_p oli ndr ome ?("a cbc a")
```

Use recursive option is less performance. It uses List.last/1 and Enum.drop/2 (O(n) each), making the complexity O(n²).

Fibonacci

```
defmodule Fibonacci do
   def fib(n), do: fib(n, 0, 1)
   defp fib(0, a, _b), do: a
   defp fib(n, a, b), do: fib(n - 1, b, a + b)
end
IO.ins pec t(F ibo nac ci.f ib (10)) # => 55
```

O(n)

Fast Enum Reference

Acumulate	reduce
Return in map or diff type than map	reduce
Return list	map
Discard / Clean	filter

Liveview callbacks

mount	
handle-_event	clicks, forms, keys, hooks.
handle-_info	messages from other processes (send)
handle-_params	Respond to URL changes

Actor model

The actor model is a concurrency paradigm in which **processes (called actors)** communicate exclusively through message passing, **without sharing memory.**
The actor model uses a **FIFO queue-like** structure to process messages, but encapsulated in each actor.
It is not a traditional queue because each actor is an autonomous and concurrent system.

GenServer Callback

init	
handle_call	
handle_cast	
handle_info	
code_c-hange	Hot code update
terminate	

Best for inserts / updates

Lists (Prepend)	[new_item | list]
Maps insert	Map.put O(1) except hash collisions
Maps update	%{map | a: 42} better than Map.put
Keyword List (Prepend)	[{:key, val} | kw_list]

Process Communication Elixir/Erlang

Interprocess communication in Erlang and Elixir is based on the **actor model** and is completely **asynchronous**. Each process is lightweight, running on the BEAM virtual machine, and has its **own state** and **message queue**. Processes **do not share memory**, allowing for **lock-free concurrency** and fault tolerance.
1-Erlang cookie is same in 2 nodes
2-Node1 ask for the Node2 port using **EPMD**

Process Communication Elixir/Erlang (cont)

3-Node 1 send a message to the **EDP** (Erlang Distribution Protocol)
Difference with process messaging in Java and why is better?
Because Java uses OS threads, these can share memory, every thread use stack and OS resources.

Strings to charlist

```
#Directly using sigil ~c
~c"a bc"
#from symbol
def foo(name) do
    to_ cha rli st( -
name)
end
```

Imports, alias, use, require

use	Extend modules and add functi-onality	use GenServer
require	Use macros from another module	require Logger
import	Use functions without module prefix	import Enum, only: [map: 2]
alias	Short modules names	alias MyApp.F-oo.Foo2

Difference GenServer VS GenState

```
GenServer has state and
response to casts.
GenState is data-flow
with backpr esure.
```

String to List ("AB"=>["A", "B"])

```
String.gr aph eme s("A -
B")
```

Interfaces with Elixir (polymorp-hism)

first defines the **@callbacks** in an interface module...
Then implement in a module using **@behavior**
Lastly, when we "override" a function defined in interface or which has a **callback** use `@impl true`

inheritance in Elixir

The most similar to "inheritance" is to use the `use` macro. This extends functions in the module.

OBAN vs EXq

Caract erist	OBAN	Exq
DB	Postgres	Redis
Distri-buted	Multi-node	Needs Redis Cluster
Retries	Config-urable with backoff	Less flexible

OBAN vs EXq (cont)

Scalable	Excelent for big systems	Good, but depends on Redis

By **zokratez**
cheatography.com/zokratez/

Published 12th March, 2025.
Last updated 3rd March, 2025.