Agenda :-

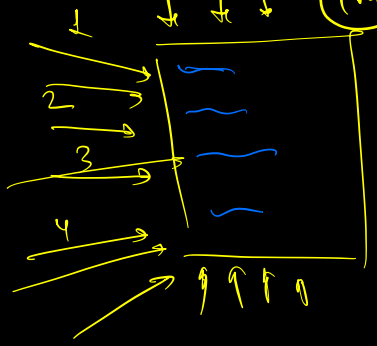
1) Executors and thread pools
   (How threads are coded in production)
2) Callables
   (How threads return data)
   → Merge sort
3) Synchronization problem
   → Adder Substractor.

1) Executors and Thread pool.

Client
- Create a task
- Decides when the task will run (in multi threaded env)

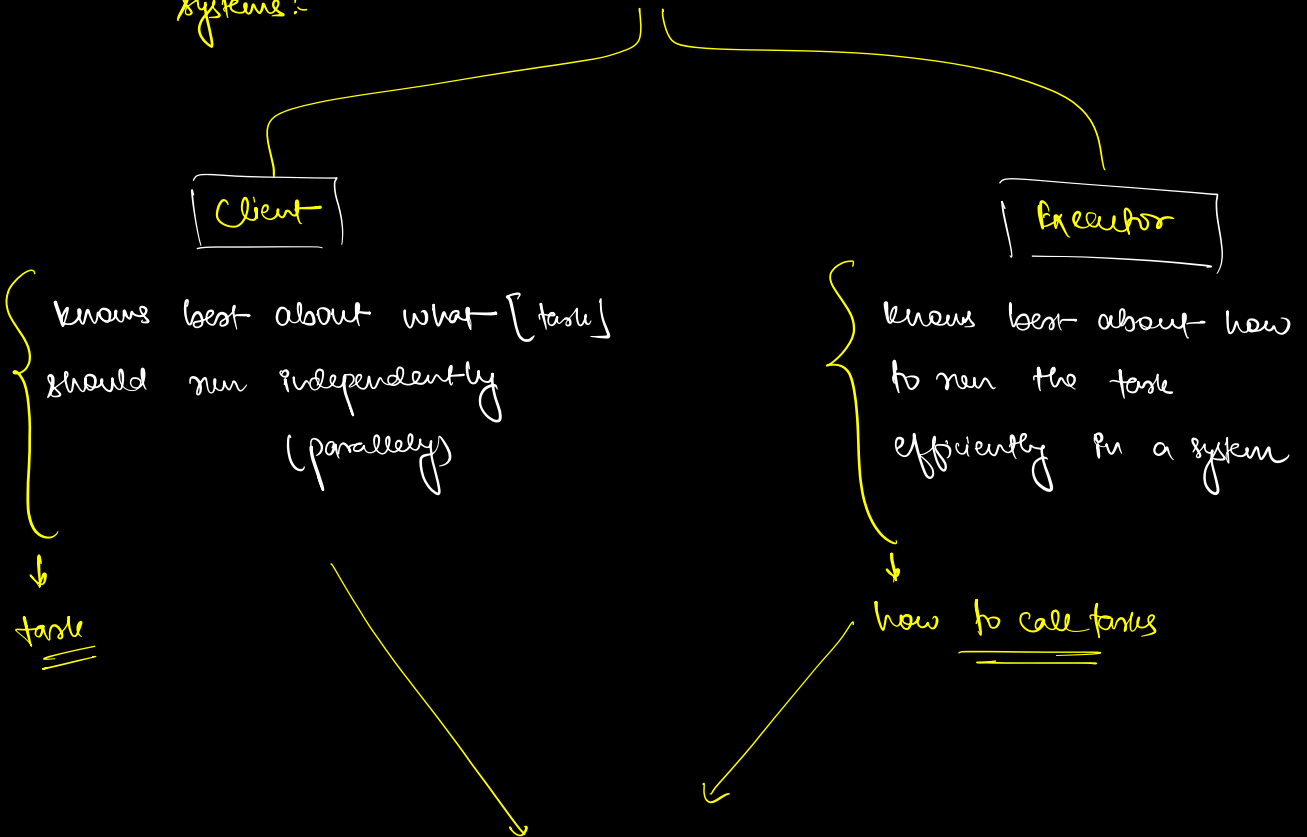
ex => Building a server :-

every request => will be one thread



=> a lot of memory usage
=> context switching

=> Divides the responsibility of multi-threaded appn into 2 systems:-



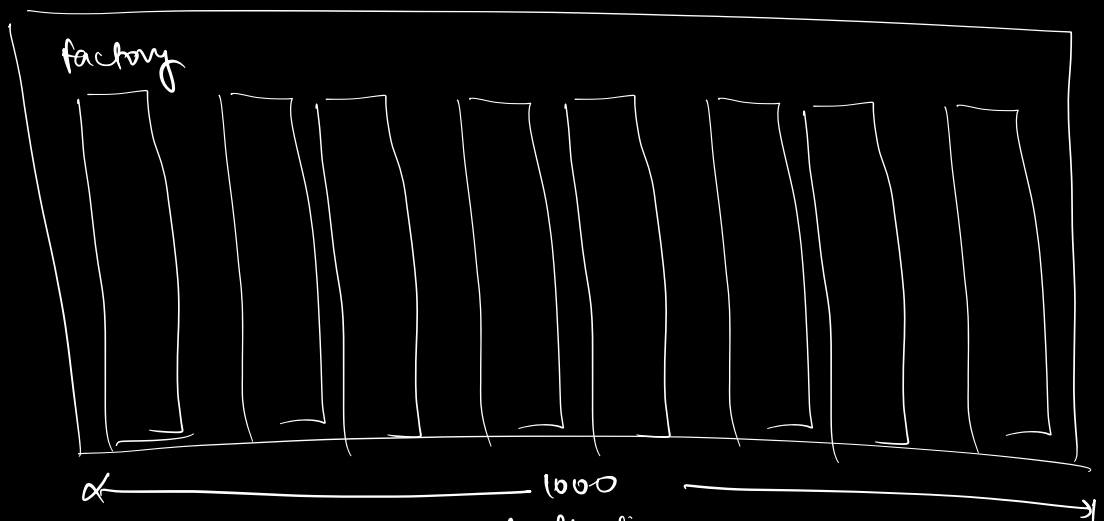
Executor framework

=> Division of responsibility

=> Efficient running of the appn

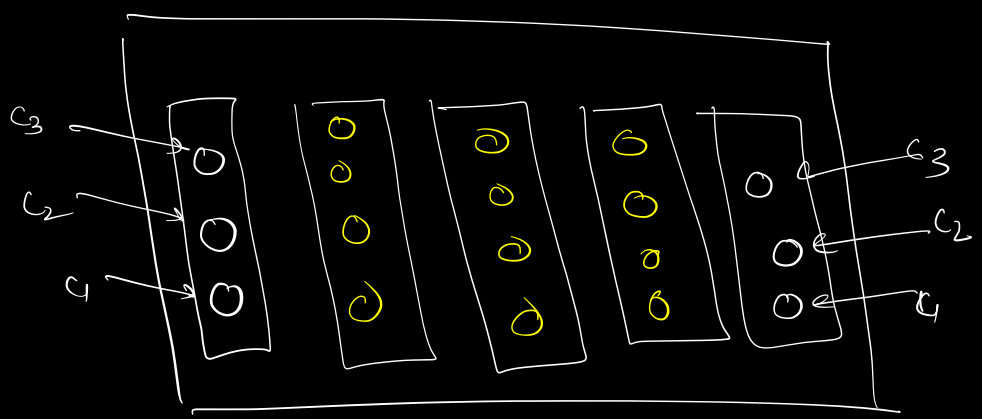
=> Executors internally use something called "Thread Pool".

⇒ Car factory [~~1000~~ 2000 cars]



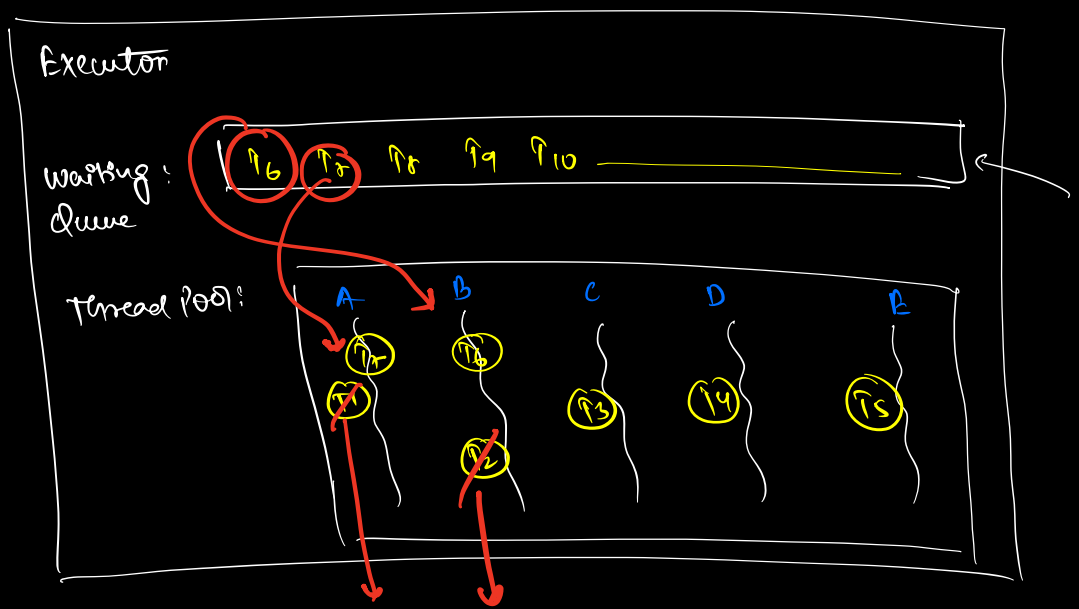
→ Not efficient

→ we are not reusing the prodn lines for future cars.



multiple cars going through each prodn line

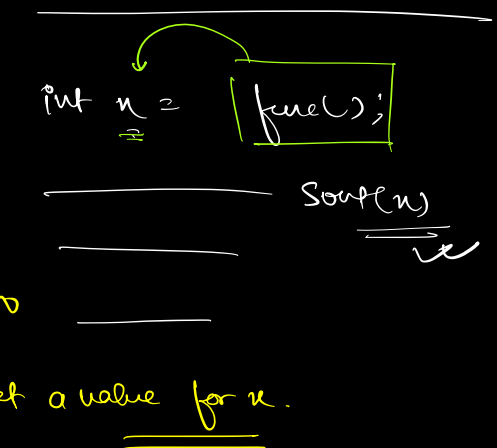
Thread Pool (set of threads acting as a producer)



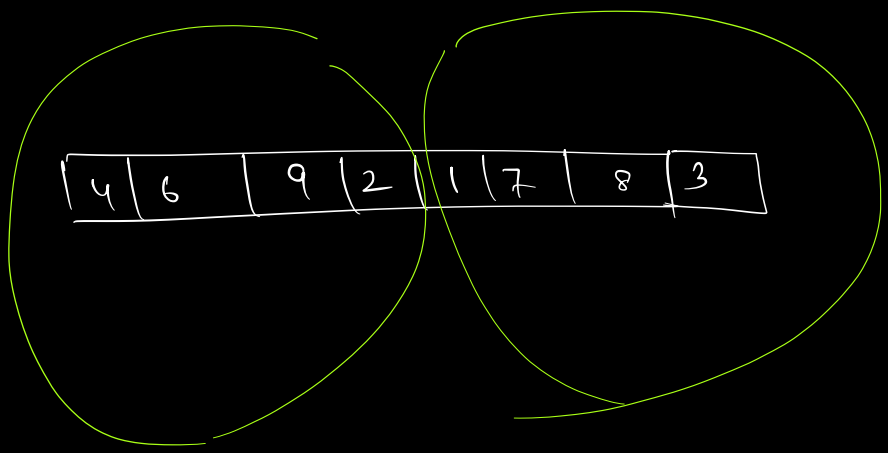
* we are not recreating threads for every new task, we are reusing them

⇒ Optimizes on thread creation and destruction time

⇒ Callables :-



Merge Sort



take a half sort it and merge with other sorted half

Q1 → sort (left)
sort (right)
] → independent tasks

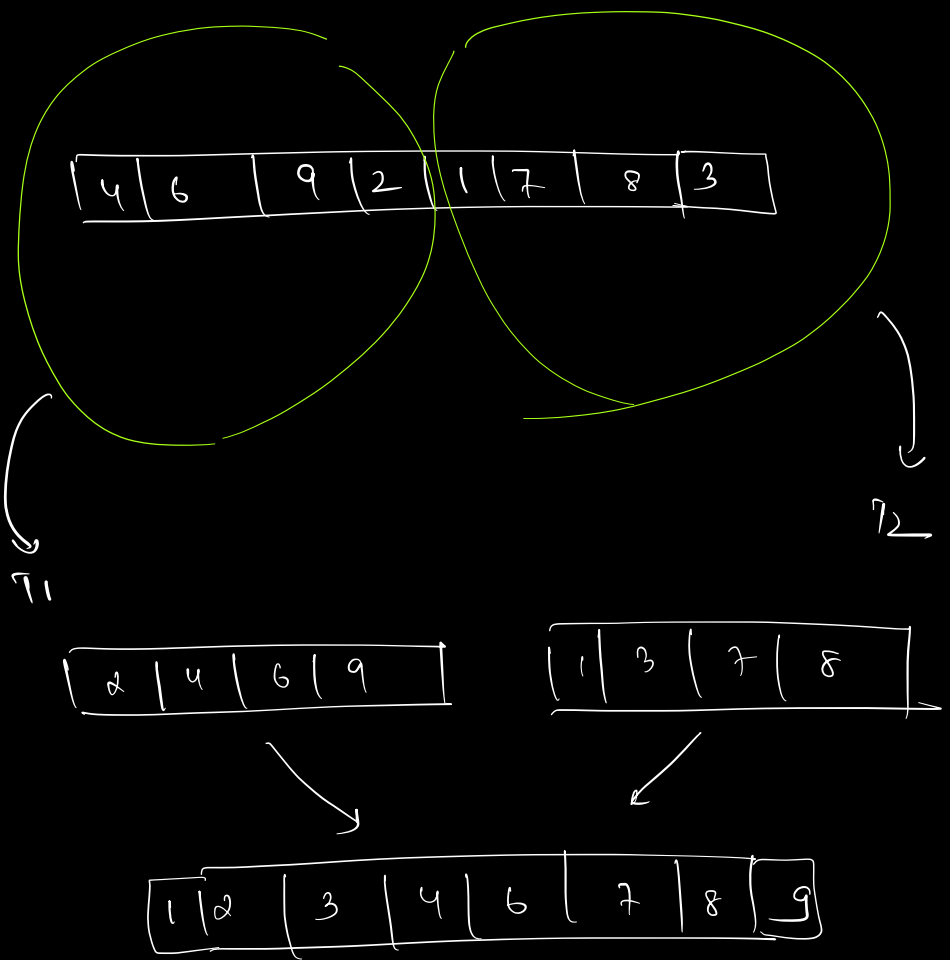
merge(sortedL, sortedR)

↓

merge func needs both sorted half, to execute, thus dependent on return value of both threads, sorting each half

↓

Can be done in 2 separate threads Parallely



Callables :- => Like Runnable, Callables are a way to define a task

=> Unlike Runnable, Callables return something back to the client

1) Identify the tasks to be run in parallel thread. Create a class for that :-

=> name of runnable / callable should be a noun

=>

class Sorter { }

(T)

1) Identify the return type of data that the task will return. T => List<Integer>.

2) make the task class implement Callable<T>

3) Implement a method call() { }

Generics

Assign-1 — Read and code about generics

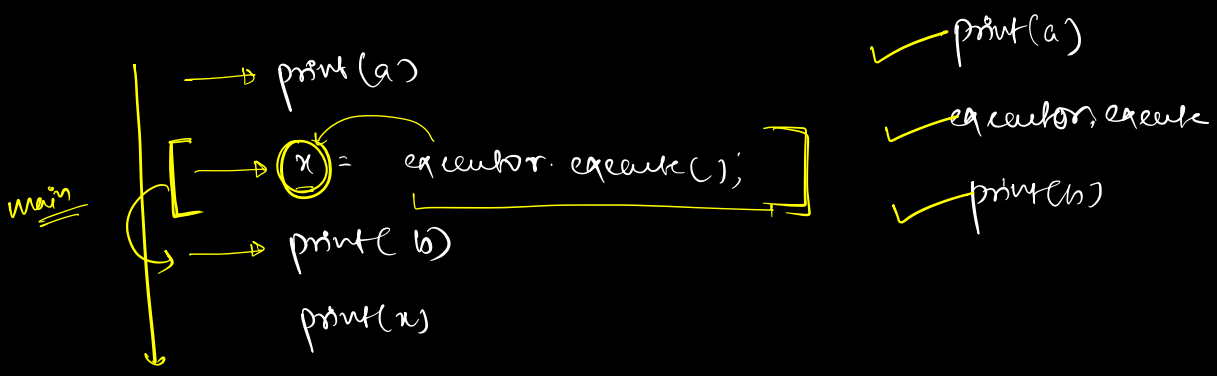
class Sorter implement Callable<List<Integer>> {

List<Integers> call() {

call() ← execute call to have recursion

}

: FUTURE



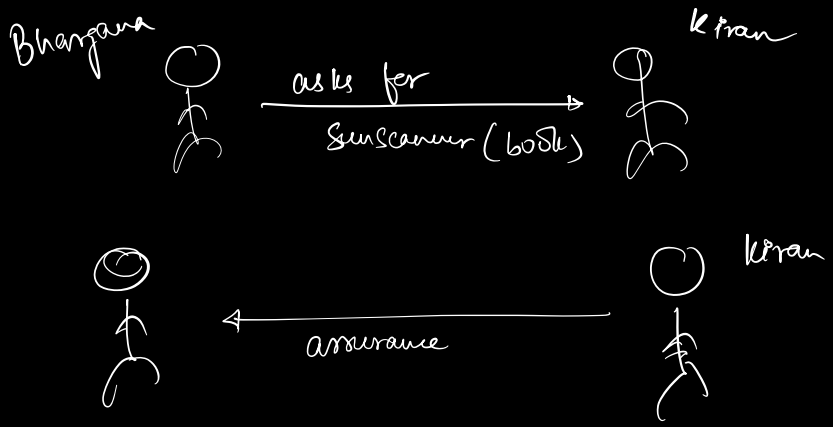
executor.submit() → instead of returning the real data.
it returns a future instead.

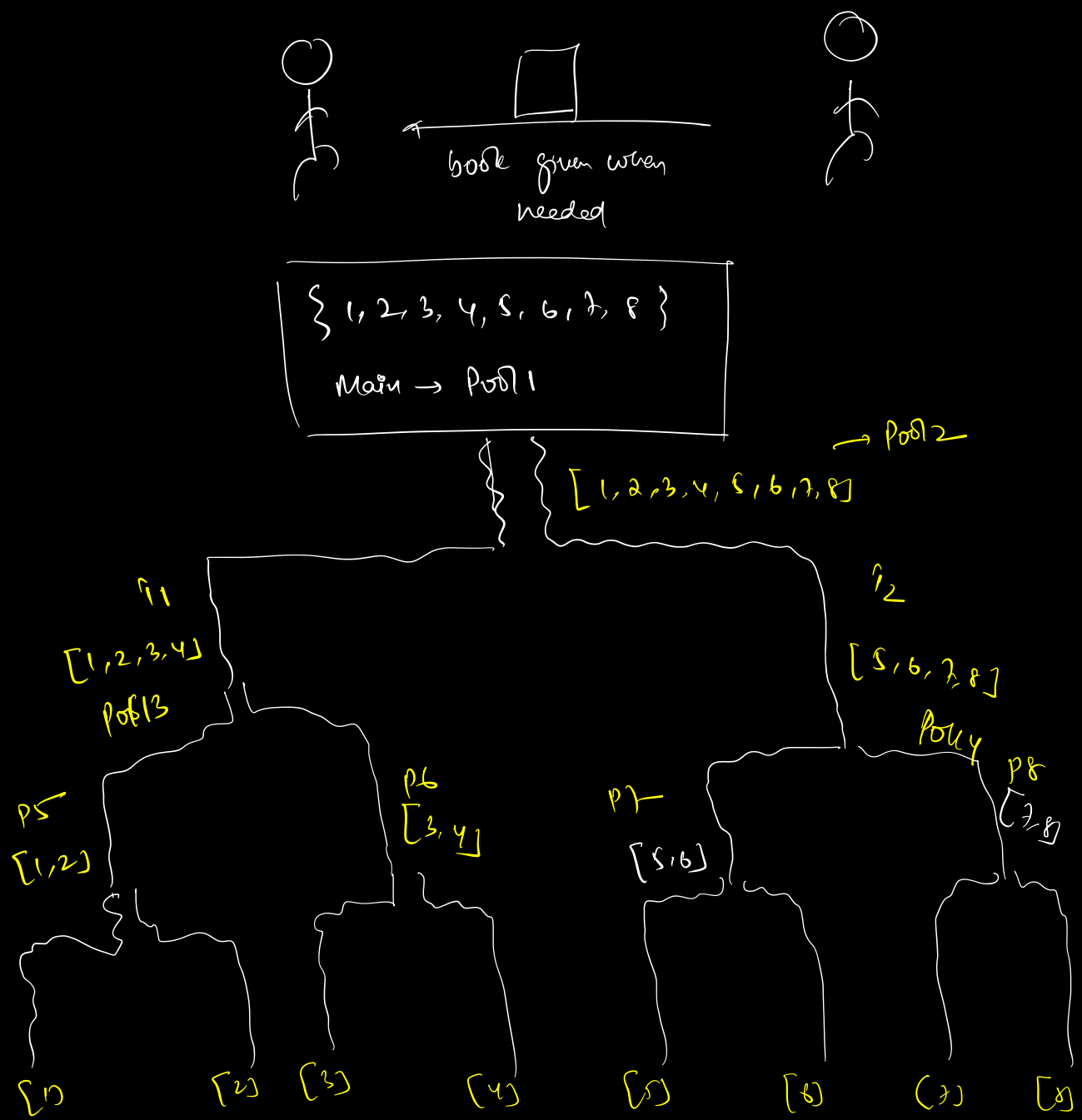
↓

future< Datatype >

{ eg ⇒ instead List<Integer>, it returns
future<List<Integer>>

future ⇒ a bucket (assurance) to give the real data when required.




book given when needed
{1, 2, 3, 4, 5, 6, 7, 8}
Main → Pool1
→ Pool2
[1, 2, 3, 4, 5, 6, 7, 8]
f1
[1, 2, 3, 4]
Pool3
f2
[5, 6, 7, 8]
Pool4
P5
[1, 2]
P6
[3, 4]
P7
[5, 6]
P8
[7, 8]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]