```
1. Find the last box of a list.
e.g.
    p01([1,2,3,4])
    > 4

defmodule Solution do
  def start([head | tail]) do
    solve(head, tail)
  end

  def solve(old_head , [head | tail]) do
    new_head = head
    solve(new_head, tail)
  end

  def solve(last_head, empty_tail) do
    last_element = last_head
    IO.puts(last_element)
  end
end

Solution.start([1,2,3,4,5])

2. Find the last but one box of a list.
e.g.
    p02([a,b,c,d,e,f,g])
    > [f,g]

defmodule Solution do
  def start([head | tail]) do
    solve(nil, head, tail)
  end

  def solve(last_one, last_two, [new_head | tail]) do
    last_one = last_two
    last_two = new_head
    solve(last_one, last_two, tail)
  end

  def solve(last_one, last_two, empty_tail) do
    [last_one, last_two]
  end
end

Solution.start([1,2,3,4,5,6,7])


3. Find the K'th element of a list
e.g.
    p03([1,2,3,f,b,3,f,s], 4)
    > f

defmodule Solution do
  def start([head | tail], num_of_element) do
```

```
    counter = 1
    solve(counter, num_of_element, head, tail)
  end

  def solve(counter, num_of_element, head, [new_head | tail]) when
counter == num_of_element, do: head

  def solve(counter, num_of_element, head, [new_head | tail]) do
    counter = counter + 1
    head = new_head
    solve(counter, num_of_element, head, tail)
  end

  def solve(counter, num_of_element, head, end_of_list), do: head
end

Solution.start([:a, :b, :c, :d, :e, :f, :g], 7)
```

4. Find the number of elements of a list
e.g.
```
    p04([1,2,x,3,4,5])
    > 6
```

```
defmodule Solution do
  def start([_head | tail]) do
    num_of_elements = 1
    solve(num_of_elements, tail)
  end

  def solve(num_of_elements, [_head | tail]) do
    num_of_elements = num_of_elements + 1
    solve(num_of_elements, tail)
  end

  def solve(num_of_elements, end_of_list), do: num_of_elements
end

Solution.start([:a, :b, :c, :d, :e, :f, :g])
```

5. Reverse a list
e.g.
```
    p05([1,z,2,3,4])
    > [4,3,2,z,1]
```

```
defmodule Solution do
  def start([head | tail]) do
    reverse_list = [head]
    solve(reverse_list, tail)
  end

  def solve(reverse_list, [head | list]) do
    reverse_list = [head | reverse_list]
    solve(reverse_list, list)
  end

  def solve(reverse_list, end_of_list), do: reverse_list
end

Solution.start([:a, :b, :c, :d, :e, :f, :g])
```

6. Find out whether a list is a palindrome.
e.g.

```
    p06([1,2,3,2,1])
    > true
    p06([1,2,3,3,1])
    > false
```

```
defmodule Solution do
  def start(list) do
    reverse_list = Enum.reverse(list)
    solve(list, reverse_list)
  end

  def solve([head | list], [head_reverse | reverse_list]) do
    if head === head_reverse do
      solve(list, reverse_list)
    else
      false
    end
  end

  def solve(list, reverse_list), do: true
end

Solution.start([1, 2, 3, 2, 1])
```

7. Flatten a nested list structure
e.g.

```
    p07([a,[b,[c,d],e]])
    > [a,b,c,d,e]
List.flatten([:a,[:b,[:c,:d],:e]])
```

```
defmodule Solution do
  def start(list) do
    solve(list, [])
  end

  def solve([head | list], result) do
    solve(head, list, result)
  end

  def solve([new_head | new_list], list, result) do
    solve(new_head, new_list, result)
  end

  def solve(element, list, result) do
```

```
    result ++ [element]
    solve(list, result)
  end

  def solve(element, list, result) when result === [] do
    result ++ [element]
    result
  end
end

Solution.start([:a,[:b,[:c,:d],:e]])
```

8. Eliminate consecutive duplicates of list elements.
e.g.

```
    p08([a,a,a,a,b,c,c,a,a,d,e,e,e,e)
    > [a,b,c,d,e]
Enum.uniq([:a,:a,:a,:a,:b,:c,:c,:a,:a,:d,:e,:e,:e,:e])
defmodule Test do
  def start(list) do
    solve(list, [])
  end

  def solve([], uniq_list), do: uniq_list
  def solve([attr_to_check | list_to_check], uniq_list) do
    checker = uniq_checker(attr_to_check, uniq_list)

    if checker do
      new_uniq_list = uniq_list ++ [attr_to_check]
      solve(list_to_check, new_uniq_list)
    else
      solve(list_to_check, uniq_list)
    end
  end

  def uniq_checker(_attr_to_check, uniq_list) when uniq_list == [], 
do: true
  def uniq_checker(attr_to_check, [uniq_value | uniq_tail]) do
    if uniq_value === attr_to_check do
      false
    else
      uniq_checker(attr_to_check, uniq_tail)
    end
  end
end

Test.start([:a,:a,:a,:a,:b,:c,:c,:a,:a,:d,:e,:e,:e,:e])
```

9. Pack consecutive duplicates of list elements into sublists.
e.g.

```
    p09([a,a,a,a,b,c,c,a,a,d,e,e,e,e])
    > [[a,a,a,a],[b],[c,c],[a,a],[d],[e,e,e,e]]
defmodule Test do

  def p09(x), do: p(x, [])
  def p([], acc), do: acc
  def p([h | t], acc) do
    new_acc = present(acc, acc, h)
    p(t, new_acc)
  end

  def present([], acc, h), do: [[h] | acc]
  def present([[l | _] = hacc | tacc], acc, h) when l == h do
    newhacc = hacc ++ [h]
    acc -- hacc ++ [newhacc]
  end
  def present([_ | tacc], acc, h) do
    present(tacc, acc, h)
  end
end

Test.p09([[:a,:a,:a,:a],[:b],[:c,:c],[:a,:a],[:d],[:e,:e,:e,:e]])


10. Run-length encoding of a list.
e.g.
    p10([a,a,a,a,b,c,c,a,a,d,e,e,e,e])
    > [[a,4],[b,1],[c,2],[a,2],[d,1],[e,4]]
#!/usr/bin/env elixir
defmodule Test do
  def start(list) do
    solve(list, [], nil, 1)
  end

  def solve([attr_to_check | list_to_check], result_list, nil,
counter) do
    solve(list_to_check, result_list, attr_to_check, counter)
  end
  def solve([attr_to_check | list_to_check], result_list,
prev_value, counter) do
    is_duplicate = duplicates_checker(attr_to_check, prev_value)

    if is_duplicate do
      counter = counter + 1
      solve(list_to_check, result_list, attr_to_check, counter)
    else
      new_result_list = result_list ++ [[prev_value, counter]]
      solve(list_to_check, new_result_list, attr_to_check, 1)
    end
  end
  def solve([], result_list, prev_value, counter) do
    result_list ++ [[prev_value, counter]]
  end

  def duplicates_checker(attr_to_check, prev_value) do
    if attr_to_check === prev_value do
      true
    else
```

```
      false
    end
  end
end

Test.start([:a,:a,:a,:a,:b,:c,:c,:a,:a,:d,:e,:e,:e,:e])
```