Dynamic Programming Ruby

Dynamic Programing : a class / type of algorithm, can be used to solve certain types of problems.

2 main properties for problems to be solved in dynamic programming approach:
- Optimal Substructure
- Overlapping Subproblems

4 steps

1) Recurrence Relation : a relation that repeats itself. (~Recursion)
2) Top down : Solve problem ~~from~~ a global solution, then breaking it down into smaller ones, finding solutions to each of them until we got a global one.
3) Bottom up : find solutions of all small sub problems then build a global one.
4) Optimize : find ways to optimize the btm-up solution so that we have a more efficient solution in terms of memory & runtime performance.

↳ steps ① & ② to check if a problem has an optimal substructure or overlapping sub problems.

- Recursion

↳ Recursive function : A function that calls itself until it hits a stopping condition (base case).

↳ 2 main examples : Factorial & Fibonacci numbers (but not practical in workplace)

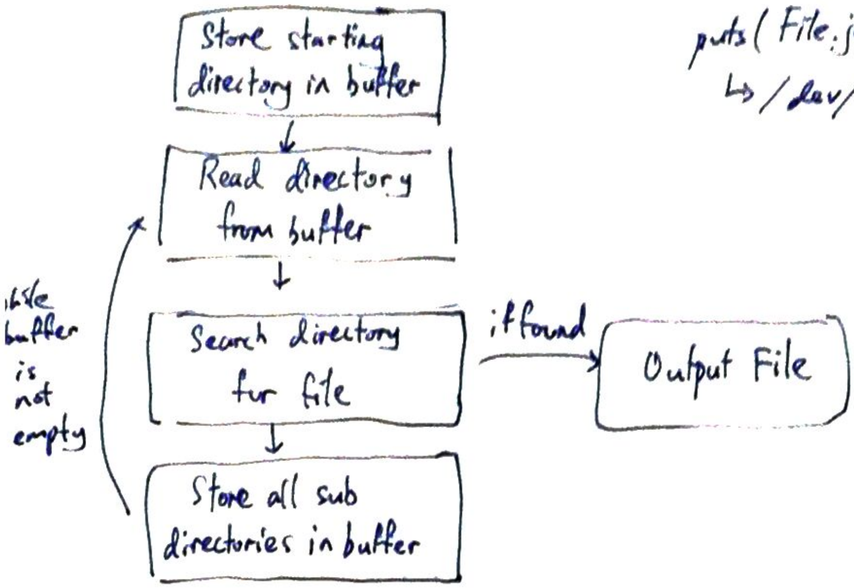
↳ 2 other practical examples : countdown timer & writing a file search in a directory.

CountDown Timer :

```ruby
def countdown(n)
  return if n<0
  puts(n)
  sleep(1)
  countdown(n-1)
end
```

Eg. Search file (readme.txt) in Folders (eg. /dev).

- non-recursive manner :

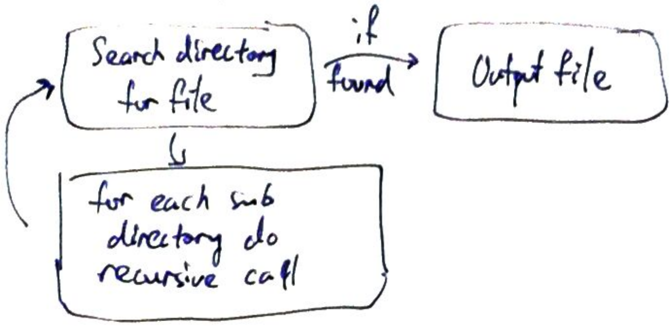


- Join 2 string together "/dev/readme.txt by

puts(File.join ("/dev", "readme.txt"))

↳ /dev/readme.txt

- Recursive Approach :



5) Counting Derangements

6) Coming up with a Recurrence Relation

- Try to express solution recursively
- Check whether ur function satisfy the optimal substructure condition
- Start from base, move to general

7) Recursive code walkthrough

but performing poorly, doesn't scale very well for large inputs.

```
class CountDerangementsRec
    def initialize(set_size)
      @set_size = size_size
    end
    def count_derangements (n =@set_size)
      if n==1
        0
      elseif n==2
        1
      else
        (n-1)* (count_derangements (n-1) + count_derangements(n-2))
      end
    end
  for i in 1..64
    result = CountDerangementsRec.new(i).count_derangements
    puts("Derangements in set size %d -> %d" % [i, result])
  end
```

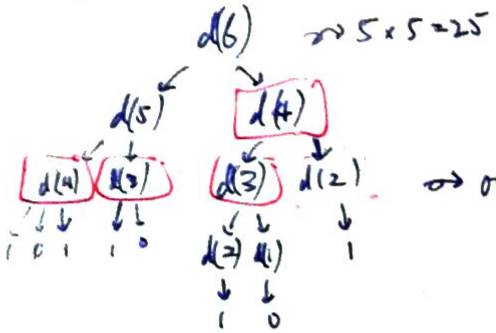
8) Top Down solution → solving dynamic programming using temporary space

$$d(n) = \begin{cases} 0 & \text{if } n == 1 \\ 1 & \text{if } n == 2 \\ (n-1)(d(n-1) + d(n-2)) \end{cases}$$

→ to use a diff. algorithm so we can optimize that & compute the current arrangements in a much more efficient manner.

- Draw a tree of all diff. recursive calls that we had in brute force algo.

eg. If brute force mthd:



→ overlapping subproblems

To Use memoization : once we store solution, it never times out, use it whenever we need it, it's in our memory space.

d6

← n+1 →

[-1, -1, -1, -1, 9, -1, -1] → each position represents the solution for the input corresponding to its index.

0 1 2 3 4 5 6

d(4) = 9

9) Top-down code walkthrough

```
class CountDerangementsTopDown
  def initialize(set_size)
    @set_size = set_size
    @sub_solutions = Array.new(set_size+1, -1)   → initialize with -1
  end
  def count_derangements(n=@set_size)
    if @sub_solutions[n] != -1
      @sub_solutions[n]
    elsif n==1
      0
    elsif n==2
      1
    elsif
      result = (n+1)*(count_derangements(n-1) + count_derangements(n-2))
      @sub_solutions[n] = result
      result
end end end
```

```
for i in 1..64
  n= CountDerangementsTopDown.new(i).count_derangements
  puts("Derangements in set size %d -> %d" % [i,n])
end
```

10) Bottom up Solution.

d(6)

<——— n+1 ———>

[-1, 0, 1, 2, 9, 44, 265]

0 1 2 3 4 5 6

→ → → 2(1+0) 3(2+1) 4(9+2) 5(44+9)

* need a space to store sub-solutions.

11) Btm up Code Walkthrough

```
class CountDerangementsBottomUp
  def initialize(set_size)
    @set_size = set_size
    @sub_solutions = Array.new(set_size + 1)
    for n in 1..@set_size
      if n==1
        @sub_solutions[n] = 0
      elsif n==2
        @sub_solutions[n] = 1
      else
        @sub_solutions[n] = (n - 1)*(@sub_solutions[n+1] + @sub_solutions[n-2])
      end
    end
  end
  def count_derangements
    @sub_solutions[@set_size]
  end
end
for i in 1..64
  n= CountDerangementsBottomUp.new(i).count_derangements
  puts("Derangements in set size %d -> %d" % [i,n])
end
```

12) Optimization & code walkthrough

```
class CountDerangementsOpt
  def initialize(set_size)
    @set_size = set_size
    @solution_n = solution_n_minus_1 = solution_n_minus_2 = 0
    for n in 1..@set_size
      if n ==1
        @solution_n = 0
      elsif n ==2
        @solution_n = 1
      else
        @solution_n = (n-1)*(solution_n_minus_1+ solution_n_minus_2)
      end
      solution_n_minus_2 =solution_n_minus_1
      solution_n_minus_1 = @solutin_n
    end
  end
  def count_derangements
    @solution_n
  end
end
for i in 1..64
    n = CountDerangementsOpt.new(i).count_derangements
    puts("Derangements in set_size %d -> %d" %[i, n])
end
```

13) Solving Air Traffic

- Given 7 aircrafts with diff. # of passengers, ~~rule: try to~~
- Rule: Try to land as many passengers as possible ; No two adjacent aircrafts can land after each other.

~~tof~~ Cus first come first serve doesn't give the optimum solution.

14) Defining a solution recursively [155, 55, 2, 96, 67, 203, 3]

- Recurrence Relation
  - base case: max passengers = 0 if list == []

$$\max \begin{cases} 155 + \text{maxPass}([2, 96, 67, 203, 3]) \\ \text{maxPass}([55, 2, 96, 67, 203, 3]) \end{cases}$$

$$\text{maxPass}(0) = \max \begin{cases} @0 + \text{maxPass}(2) \quad \{0 \text{ if list} == [] \\ \text{maxPass}(1) \end{cases}$$

input length

$$\text{maxPass}(i) = \max \begin{cases} @i + \text{maxPass}(i+2) \quad \{0 \text{ if list} == [] \rightsquigarrow 0 \text{ if } i \geq |\text{Input}| \\ \text{maxPass}(i+1) \end{cases}$$

```ruby
class AircraftSpacingRec
  def initialize(passengers)
    @passengers = passengers
  end

  def max_passengers(i)
    if i >= @passengers.length
      0
    else
      choosing_first = @passengers[i] + max_passengers(i+2)
      not_choosing_first = max_passengers(i+1)
      [choosing_first, not_choosing_first].max
    end
  end
end
```

```ruby
spacing = AircraftSpacingRec.new([155,5,2,96,67,203,3])
puts(spacing.max_passengers(0))
```

first item of the list

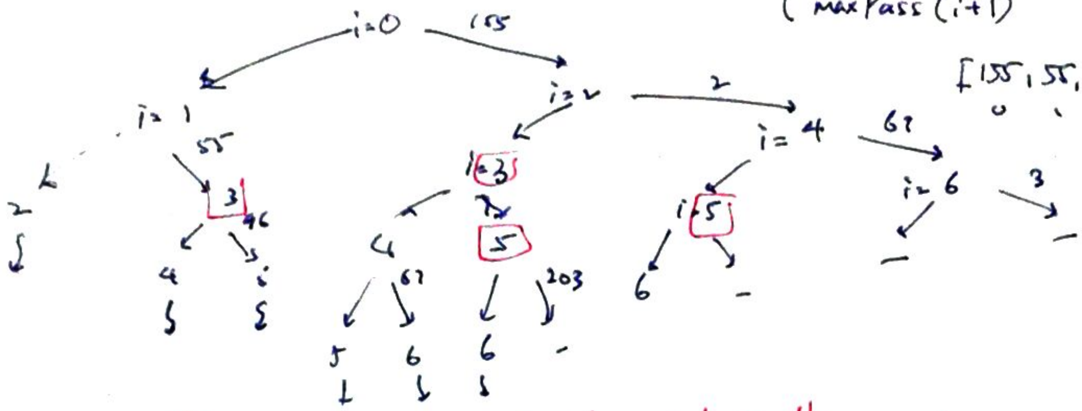
↳ 454 ✓

↳ slow on big sample ; doesn't scalable.

16) Top Down Solution.

$$\max \begin{cases} 0 \text{ if } i \geq l \text{ input } l \\ @i + maxPass(i+2) \\ maxPass(i+1) \end{cases}$$

[155, 55, 2, 96, 67, 203, 3]
0 1 2 3 4 5 6



↳ use memo-ization to store sub-results, so when you encounter agn, you don't need to re-compute.

↳ By initializing an array of -1 with same size of inputs, i.e.

[-1, -1, -1, -1, -1, -1, -1] → indicate the sub-solution is empty.
0 1 2 3 4 5 6

4 ↓ 203

17) Top Down code - walkthrough.

```
class AircraftSpacingTopDown
  def initialize(passengers)
    @passengers = passengers
    @sub_solutions = Array.new(passengers.length, -1)
  end
  def max_passengers(i)
    if i >= @passengers.length
      0
    elsif @sub-solutions[i] != -1
      @sub_solutions[i]
    else
      choosing_first = @passengers[i] + max_passengers(i+2)
      not_choosing_first = max_passengers(i+1)
      max_pass = [choosing_first, not_choosing_first].max
      @sub-solutions[i] = max_pass
      max_pass
    end
  end
end
```

↳ spacing = AircraftSpacingTopDown.new([155, 55, 2, 96, 67, 203, 3])
puts(spacing.max_passengers(0))

✓

18) Btm Up Approach

↳ Involves filling up memoization space with pre-computed values instead of filling them up as you go. We can fill it up before calling.

$$f(i) = \begin{cases} 0 \text{ if } i \geq \text{input } l \\ \max \begin{cases} @i + maxPass(i+2) \\ maxPass(i+1) \end{cases} \end{cases}$$

[-1, -1, -1, -1, -1, -1, 3] 0 0
0 1 2 3 4 5 6 7 8

↑ because 6th can be computed just from base cases 7 & 8.

$$= \max \begin{cases} @i + maxPass(i+2) \\ maxPass(i+1) \end{cases}$$

$$= \max \begin{cases} 3 + maxPass(8) \\ maxPass(7) \end{cases} = \max \begin{cases} 3+0 \\ 0 \end{cases} = 3.$$

At 5th,

$$f(5) = \max \begin{cases} @5 + maxPass(7) \\ maxPass(6) \end{cases}$$

$$= \max \begin{cases} 203 + 0 \\ 3 \end{cases} \rightarrow 203.$$

$$f(4) = \max \begin{cases} 67 + 3 \\ 203 \end{cases} \rightarrow 203$$

$$f(3) = \max \begin{cases} 96 + 203 \\ 203 \end{cases} \rightarrow 299$$

$$f(2) = \max \begin{cases} 2 + 203 \\ 299 \end{cases} \rightarrow 299$$

$$f(1) = \max \begin{cases} 55 + 299 \\ 299 \end{cases} \rightarrow 345$$

$$f(0) = \max \begin{cases} 155 + 299 \\ 345 \end{cases} \rightarrow \underline{454}$$

[454, 345, 299, 299, 203, 203, 3]

19) Btm Up code walkthrough → gives us a glimpse into any possible optimization.

```
class AircraftSpacingBottomUp
  def initialize(passengers)
    @passengers = passengers
    @sub_solutions = Array.new(passengers.length - 1)
    for i in (@passengers.length - 1).downto(0)
      choosing_first = @passengers[i] + (i+2 < @sub_solutions.length ? @sub_solutions[i+2] : 0)
      not_choosing_first = i + 1 < @sub_solutions.length ? @sub_solutions[i+1] : 0
      @sub_solutions[i] = [choosing_first, not_choosing_first].max
    end
  end
  def max_passengers
    @sub_solutions[0]
  end
end
spacing = AircraftSpacingBottomUp.new([155, 55, 2, 96, 67, 203, 3])
puts(spacing.max_passengers) → 454
```

20) Cs drawback : consumes more memory than what's needed. (eg. for large inputs it need large area to store these substitutions.)

eg. for item 0, only require item 2 & 3, 4 - 6 are not needed.

Solve : replace the entire [ ] 7 8 with i, i+1, i+2. Initially, i i+1 i+2

i+1 & i+2 = 0

20) Optimization (Code)

```
class AircraftSpacingOpt
  def initialize(passengers)
    @passengers = passengers
    @sub_solution_i = sub_solution_i_plus_1 = sub_solution_1_plus_2 = 0
    for i in (@passengers.length - 1).downto(0)
      choosing_first = @passengers[i] + sub_solution_i_plus_2
      not_choosing_first = sub_solution_i_plus_1
     @sub_solution_i = [choosing_first, not_choosing_first].max
     sub_solution_i_plus_2 = sub_solution_i_plus_1
     sub_solution_i_plus_1 = @sub_solution_i
    end
  end

  def max_passengers
    @sub_solution_i
  end
end

spacing = AircraftSpacingOpt.new([155, 55, 2, 96, 67, 203, 3])
puts(spacing.max_passengers)
```

21) Maximum Sub Array

- How are maximum sub arrays useful?

subarray : a continuous portion of an original array ; ~~*[] is~~

*[] is sub array of every array.

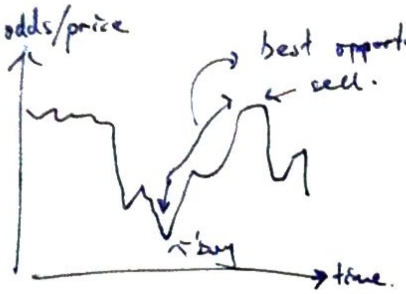
* Continuous

Maximum Sub Array :

└ Given [5, -4, 8, -10, -2, 4, -3, 2, 7, -8, 3, -5, 3], find the maximum sub array.

* It only make sense if it contains -ve & +ve numbers.

eg.



best opportunity to bet. = maximum sub array.
= max increase.
↳ by AI neural network; there is when maximum subarray goes handy

22) Recurrence Relation. (Max Sub Array)

max sub array :

eg. [2, 3, -5]

endingAt(i) - subarrays : $\underbrace{[-5], [3,-5], [2,3,-5]}_{\text{max endingAt}(2) = 0}$, $\underbrace{[3], [2,3]}_{\text{max endingAt}(1) = 5}$, $\underbrace{[2]}_{\text{max endingAt}(0) = 2}$

* include 0 ie [ ]

$$\therefore \text{max subarray} = \text{MAX}\begin{cases} \text{endingAt}(n) \\ \cdots \\ \text{endingAt}(0) \end{cases}$$

0 .. n → n+1 if 0 ... n (n is exclusive)
.. → inclusive

eg. [5, -4, 8, -10, -2, 4, -3, 2, 7, -8, 3, -5, 3]
0 1 2 3 4 5 6 7 8 9 10 11 12

$$\text{endingAt}(5) = \underset{\text{sum}}{\text{max}}\left.\begin{cases} [4] \\ [-2, 4] \\ [-10, -2, 4] \\ [8, -10, -2, 4] \\ [-4, 8, -10, -2, 4] \\ [5, -4, 8, -10, -2, 4] \end{cases}\right\} \text{endingAt}(4) + @5 = \underset{\text{sum}}{\text{max}}\begin{cases} @5 \\ @5 + \text{endingAt}(4) \end{cases}$$

$$\text{max}\begin{cases} @4 \\ @4 + \text{endingAt}(3) \end{cases} \quad \cdots \quad @0$$

$$\therefore \text{endingAt}(i) = \max \begin{cases} @0 \text{ if } i == 0 & \rightarrow \text{only if } @0 \geq 0 \\ @i \\ @i + \text{endingAt}(i-1) \end{cases}$$

23) Recursive code walkthrough - Max Sub Array.

$$[\underset{0}{\ldots}, \ldots, \underset{n}{\ldots}]$$

$$\text{maxsubarray} = \max \begin{cases} 0 \\ \text{endingAt}(n) \\ \ldots \\ \text{endingAt}(0) \end{cases}$$

```ruby
class MaxSubArrayRec
  def initialize(prices)
    @prices = prices
  end
  def max_sub_array()
    max_value = 0
    for j in 0..@prices.length-1
      max_value = [max_value, max_sub_array_ending_at(j)].max
    end
    max_value
  end
  def max_sub_array_ending_at(i)
    if i == 0
      @prices[0]
    else
      [@prices[i], max_sub_array_ending_at(i-1) + @prices[i]].max
    end
  end
end

msa = MaxSubArrayRec.new([5,-4,8,-10,-2,4,-3,2,7,-8,3,-5,3])
puts(msa.max_sub_array)
```

In Ruby, loop is inclusive, $\therefore -1$

$$\rightarrow \text{endingAt}(i) = \max \begin{cases} @0 \text{ if } i == 0 \\ @i \\ @i + \text{endingAt}(i-1) \end{cases}$$

$$\underbrace{4, -3, 2, 7}_{10}$$

$\hookrightarrow$ 10

24) Top Down (Max Sub Array)

In initialize:

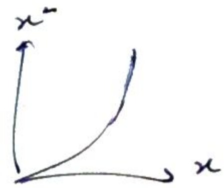
- @sub_solutions = Array.new (prices.length, nil)

In max_sub_array_ending_at(i) :

```
- if @sub_solutions[i] != nil
    @sub_solutions[i]
  elsif i == 0
    @prices[0]
  else
    m = [@prices[i], max_sub_array_ending_at(i-1) + @prices[i] ].max
    @sub_solutions [i] = m
    m
  end.
```

27) Text Justification (Typographical alignment).

- Given a chunk of words, choose which words on which lines so the paragraph looks prettier.
  - The ugly score, ↑ score = ↑ uglier, measured by spaces ^{2}




```
class TextJustifyRec
  def initialize (txt, line_length)
    @txt = txt
    @line_length = line_length
  end

  def ugly_score(txt_length)
    if txt_length <= @line_length
      (@line_length - txt_length) ** 2
    else
      Float :: INFINITY
    end
  end
end
```