NTNU | Norwegian University of Science and Technology

DYNAMIC PROGRAMMING

Jørn Vatn/August-2025

Introduction

- Dynamic programming (DP) is a method for finding an optimal strategy to apply to a dynamical system that *evolves* over *time*
- The idea for solving a dynamic programming problem is to simplify a problem into less complicated problems
- To motivate, consider the factorial function

$$f(n) = n! = \prod_{i=1}^{n} i$$

- An alternative formulation of the problem would be

$$f(n) = n \cdot f(n-1)$$

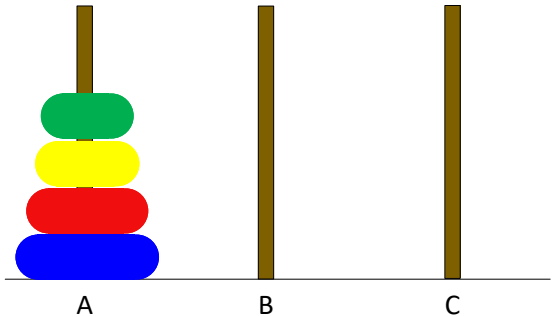
where $f(0) = f(1) = 1$

Factorial problem in Excel | Factorial problem in Python

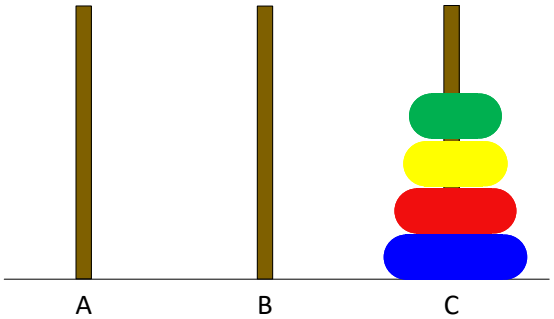
Introduction

- The idea for solving a dynamic programming problem is to split the problem in two:
  - The original problem is formulated as a sub-problem which is rather easy to solve, and then a “reduced problem”, i.e.,
    - Sub-problem: $f(n) = nf(n-1)$
    - Reduced problem: $f(n-1)$
  - The reduced problem is a simplified problem such that if we know the solution to the reduced problem, we may solve the original problem by combining the solution of the reduced problem and the sub-problem
    - If we know $f(n-1)$, we may easily solve $f(n) = nf(n-1)$
- The “reduced problem” may further be solved by the same procedure, and hopefully we at the end have a reduced problem that easily can be solved

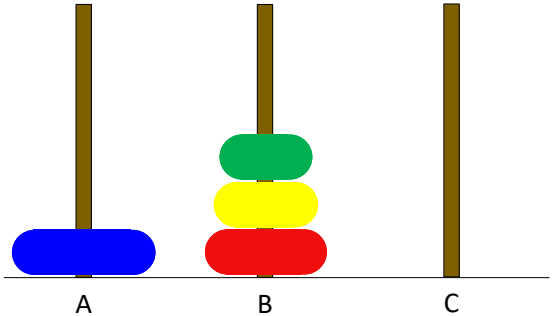
Tower of Hanoi, initial position of disks at pole A



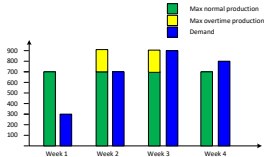
How to move the discs to pole C, small on top of large?



If we can solve this, then A → C & B → C, Done!



Recall LP problem related to production scheduling



- If we know the holding at end of week 3, say h_3, it is easy to determine production in week 4, p_4^* which minimizes $f_4(h_3, p_4)$, i.e., yielding $f_4^*(h_3)$
- Since we know $f_4(h_3, p_4)$ for all possible values of h_3, then given holding at end of week 2 is h_2, it is rather easy to minimize

$$f_3(h_2, p_3) = r_3(p_3) + f_4^*(g_3(p_3))$$

- where $r_3(p_3)$ is a return function (cost) in week 3, and $g_3(p_3)$ is a function determining the holding at end of week 3

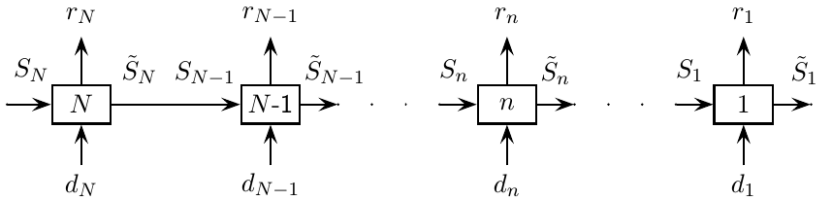
Formulating a dynamic programming problem

Let N be the number of stages a system runs through. Further the following quantities are defined for stage number n:

- $\mathbf{S}_n$ = input state, i.e., the system state when the system enters stage n
- d_n = decision at stage n, i.e., what we can do to influence the system performance
- $\tilde{\mathbf{S}}_n$ = output state, i.e., the system state when the system leaves stage n
- $r_n(\mathbf{S}_n, d_n)$ = return function, typically a cost or profit function that depends on the (input) system *state* and the *decision* made

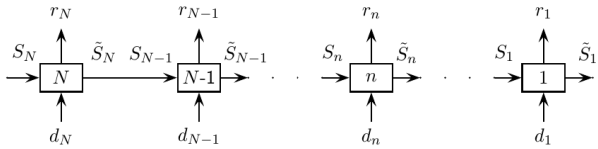
Note the difference between stage (*time* dimension) and state (system *condition* at a given stage)

Graphical illustration



The stage numbering is typically in an *opposite direction* compared to the flow of information.

Example



- We shall make decisions on how to spend totally 1 000 NOKs split for each week day Mo, Tu, We, . . ., Su
- $N = n = 7$ = Monday, $n = 6$ = Tuesday, . . . , $n = 1$ = Su
- On Sunday we spend whatever we have as leftover from Saturday, i.e., an easy problem to solve!

Solving the problem

- Start at stage 1 and proceed from right to left to stage N. Such an approach is denoted *backward analysis* or *backward recursion*
- In a backward analysis we always have that $\mathbf{S}_{n-1} = \tilde{\mathbf{S}}_n$
- The idea is that for the rightmost stage it is rather easy to find the optimal solution for a given input, i.e., for a given S_1 we easily find the best decision d_1
- When proceeding left to stage 2 we then know all the stage 1 optimal decisions and corresponding return functions r_1 for each of the output values $\tilde{\mathbf{S}}_2 = \mathbf{S}_1$
- By adding r_1 and r_2 for possible d_2 decisions, this makes it possible to find optimal decisions at stage 2 for a given input $\mathbf{S}_2$
- We then proceed till stage N is reached

Solving the problem, example

- At stage 1, i.e., Sunday, we spend the leftover from Saturday, i.e., $\mathbf{S}_{\mathrm{Su}} = \tilde{\mathbf{S}}_{\mathrm{Sa}}$
- On Sunday we spend the leftover, but the *return*, r_{Su} is not necessarily equal to $\mathbf{S}_{\mathrm{Su}}$, for example we may have $r_{\mathrm{Su}} = \ln \mathbf{S}_{\mathrm{Su}}$
- We calculate $r_{\mathrm{Su}} = r_{\mathrm{Su}}(\mathbf{S}_{\mathrm{Su}})$ for all possible values of $\mathbf{S}_{\mathrm{Su}} = \tilde{\mathbf{S}}_{\mathrm{Sa}}$
- On Saturday we decide how much to hand over to Sunday. The return on Saturday, say r_{Sa} is the *value* of what we actually spent, i.e., $\tilde{\mathbf{S}}_{\mathrm{Fr}} - \tilde{\mathbf{S}}_{\mathrm{Sa}}$
- The *objective function* to maximize on Saturday is $r_{\mathrm{Sa}} + r_{\mathrm{Su}}$ which we may obtain *if* we know the leftover from Friday
- To find that, we need to go to Friday and so on until we come to Monday, and at Monday we had $\mathbf{S}_{\mathrm{Mo}} = 1\ 000$

Transition and return functions

- The change in system state in a given stage is governed by some *transition function*, i.e.,:

$$\tilde{\mathbf{S}}_n = g_n(\mathbf{S}_n, d_n)$$

- The return function $r_n(\mathbf{S}_n, d_n)$ specifies what is the cost or profit related to what happens in stage n
- The accumulated total return calculated over n stages, i.e., from right to left or numerically from 1 to n is denoted $f_n(\mathbf{S}_n, d_n)$
- $f_n^*(\mathbf{S}_n)$ is the accumulated optimal returns up to stage n

Mathematically it is a challenge to be explicit on the formulation of $f_n(\mathbf{S}_n, d_n)$ and $f_n^*(\mathbf{S}_n)$. Typically these functions are formulated recursively.

Recursive formulation on a computer

- Let `g(n,S,d)` be a function that implements the transition that takes place in stage *n* given that decision *d* is implemented
- Let `r(n,S,d)` be a function that implements the return in stage *n* given that decision *d* is implemented
- In most cases the accumulated returns are found by adding the return function contributions
- I.e., write a recursive function, `f()`, implementing $f_n^*(\mathbf{S}_n)$ on the form:

```
f(n, S) = opt{r(n, S, d) + f(n-1, g(n,S,d))}
```

where `opt` is the optimization with respect to decision d_n at stage *n*. Let `S_N` be the initial state of the system, the optimal strategy is found by calling `f(N, S_N)`

A note on notation

- When the dynamic program problem is formulated we use the stage number n as an index, i.e., $f_n(S_n, d_n)$, $r_n(S_n, d_n)$ and $g_n(S_n, d_n)$
- When programming these functions, we cannot use n as part of the function name, since it is the same function we will implement
- We therefore pass n as a parameter, i.e., we implement functions `f(n,S,d)`, `r(n,S,d)` and `g(n,S,d)`
- Also note the distinction between $f_n(S_n, d_n)$ and $f_n^*(S_n)$. The function $f_n^*(S_n)$ returns the "best" value of $f_n(S_n, d_n)$, i.e., when $d_n = d_n^*$ = the optimal decision in stage n
- When implementing the $f()$-function in a program, we essentially implement $f_n^*(S_n)$. The syntax is `f(n,S)` without d as an argument since the function optimizes wrt d. If we decide to add d as a parameter in the function header, i.e., `f(n,S,d)`, we do this because we also would like to *pass back* the optimal value of d

Investment example

- Assume you have an initial capital of $c_\mathrm{I} = 10$ units (for example million NOKs) you are going to consume over $N = 5$ years
- Let c_t be the consumption in year t
- Note that we use c as a decision variable, rather than the standard notation d
- The utility of consuming a capital of c in one year is $u(c) = \ln c$ (not linear)
- Further, late consumption is less valuable, hence the utility is discounted by a factor $b = 0.8$ each year

Solve the investment problem with dynamic programming

Investment example, approach

- Let S be the capital going into stage n
- The accumulated utility at stage n is denoted $f_n(S) =$ `f(n, S)`
- If we spend c units of the capital in stage n, the leftover to stage $n-1$ is $S-c$
- Thus: `f(n, S) = Log(c) + b * f(n - 1, S - c)`
- Then, for stage n we maximize wrt c, i.e.,
  - `f(n, S) = max{Log(c) + b * f(n - 1, S - c)}`

Investment example, stage 2 decision

- Assume the leftover from state 3 is $\tilde{S}_3 = S_2 = 4$
- Let $f_1^*(S) = \ln(S)$ be a function that implements the best return function in stage 1
- The following calculation scheme shows the required calculations in stage 2

c	$\ln c$	$f_1^*(S_2 - c)$	$\ln c + b f_1^*(S_2 - c)$
1	0	1.1	0.88
2	0.69	0.69	1.25
3	1.1	0	1.1

Full implementation of the `f(n,S)`-function

```
Function f(n As Integer, S As Integer)
If n = 1 Then ' Consume whatever is left
  f = Log(S)
Else
  fStar = -1
  For c = 1 To S - (n - 1) ' Save minimum n-1 to subsequent stages
    fTest = Log(c) + b * f(n - 1, S - c)
    If fTest > fStar Then
      fStar = fTest
    End If
  Next c
  f = fStar
End If
End Function
```

Get the maximum utility

- In order to get the maximum utility, we can just call our `f(n,S)`-function:
  - `Debug.Print "Total utility:  " & f(5, 10)`
- This will not give us the decision at each stage
- We need “backtracking”

Backtracking

- When we call the function `f(5, 10)` it will return the optimal value, i.e., $f_5^*(S = 10)$
- Since we can only return one value in a function call, i.e., f^* we can not return the optimal c^*
- To overcome this, we add a third parameter in `f()`, i.e., we write a function `f(n, S, cStar)`
- We could then call `f(5, 10, cStar)`, and then we know that the optimal spending in stage $n = 5$ is $c^* =$ `cStar` returned from `f()`
- Next we call `f(4, 10-cStar, cStar)` to find the optimal spending in stage $n = 4$ and so on

The MainProgram

```
Sub MainProgram
S = initialCapital
n = nPeriods
Do While n > 0
  utility = f(n, S, c)  ' Get the total utility for this stage
  Debug.Print n, c, utility ' Print the intermediate results
  S = S - c ' Update state, S, based on the result from stage n
  n = n - 1 '  Go to next stage
Loop
End Sub
```

Show in Excel

Shoe store example

A shoe store sells rubber shoes for the winter season. The sales division has forecasted the demand for the next season given in below. We assume a perfect forecast, i.e., a deterministic model will be applied.

Month	Demand
October	40
November	20
December	30
January	40
February	30
March	20

Discount

The store purchases pairs of shoes from a manufacturer at a purchasing cost of 4 $ per pair (1976 prices). The supplier only sells in lots of 10, 20, 30, 40 or 50 pair. A discount is achieved for large orders:

Lot Size	Discount
0	0%
10	5%
20	5%
30	10%
40	20%
50	25%

Other costs and constraints

For each order there is a fixed cost of 10 $ to account for transportation, insurance, packaging, and so on. The shop has limited capacity and can not carry an inventory level higher than 40 at the end of each month. There is a holding cost of 0.2 $ based on the end-of-month inventory level. The season starts with an empty inventory, and should also end with an empty inventory:

Parameter	Value	Description
c_{P}	4	Purchasing cost per unit
c_{F}	10	Fixed cost per order
c_{H}	0.2	Holding cost per unit at end of month
SL	40	Storage limitation at end of month
S_6	0	Inventory level at the beginning of the season
$S_0 = \tilde{S}_1$	0	Inventory level at the end of the season

Qualitative analysis and problem structuring

The cost structure favours large orders, but the downside is a holding cost at the end of each month. Also large order could save the fixed cost if we could escape from ordering one month. It is assumed that the pair of shoes arrives the first day of the month.

- Stages: October corresponds to $n = 6$, November to $n = 5$ and so on down to March where $n = 1$
- d_n = the decision variables, are the number of pairs to order for each month
- S_n = the number of pairs in the beginning of the month when the order from the manufacturer has arrived
- D_n = Demand each month

Transition and return functions

The transition function must connect the state variables at stage n and $n-1$:

$$S_{n-1} = S_n + d_n - D_n \tag{1}$$

Return function:

$$r_n(S_n, d_n) = \phi(d_n) + c_{\mathrm{H}}(S_n + d_n - D_n) \tag{2}$$

- $\phi(d_n)$ is the cost of an order comprising the fixed cost and the discounted variable cost per pair of shoes, $\phi(0) = 0$
- Assume that the transition function in Equation (**??**) and the return function in Equation (**??**) are implemented by the functions `g(n,S,d)` and `r(n,S,d)` respectively

Recursive formulation

```
Function f(n As Integer, S As Integer)
If n = 0 Then
    f = 0
    Exit Function
End If
fOptimum = infty
For Each d In lotSizes
     fTest = r(n, S, d) + f(n - 1, g(n, S, d))
     If fTest < fOptimum Then
         fOptimum = fTest
     End If
Next
f = fOptimum
End Function
```

Recursive formulation, check for feasible solution

We need a check to ensure we have a feasible solution:

```
:
For Each d In lotSizes
    isFeasible = True
    If (g(n, S, d) > 40 Or g(n, S, d) < 0) Then isFeasible = False
    If (n = 1 And g(n, S, d) <> 0) Then isFeasible = False
    If isFeasible Then
        fTest = r(n, S, d) + f(n - 1, g(n, S, d))
        If fTest < fOptimum Then
            fOptimum = fTest
        End If
    End If
Next
:
```

Memoization

- These results for $n = 1$ are needed several times when the function is called from stage $n = 2$. Therefore it would be a good idea to save the optimal values. This applies also for all other sates
- For this example we need to calculate $f_n()$ more than 1 100 times, which reduces to around 30 calculations by clever saving of intermediate results
- Memoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again
- Memoization here means to save calculated values for f^* for (S, n) combinations

Show in Excel

When is the problem simple enough to write the solution directly?

- The recursive formulation
  - `f(n, S) = opt{r(n, S, d) + f(n-1, g(n,S,d))}`
- Will “dig” into the problem until we have to solve `f(1,S)` or sometimes `f(0,S)`, where `S` is the state when entering stage $n = 1$ or $n = 0$
- In the winter shoe problem we returned the value `f(0,S)=0` corresponding to no “return” in April. In principle we should make a check on S, and the only legal value would in this case be $S = 0$

Thank you for your attention