WIKIPEDIA
The Free Encyclopedia

Dynamic programming

Dynamic programming is both a mathematical optimization method and an algorithmic paradigm. The method was developed by Richard Bellman in the 1950s and has found applications in numerous fields, from aerospace engineering to economics.

In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively. Likewise, in computer science, if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have *optimal substructure*.

If sub-problems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the sub-problems.[1] In the optimization literature this relationship is called the Bellman equation.

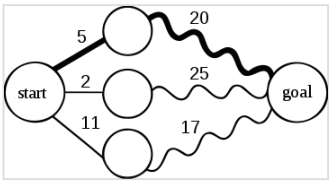


Figure 1. Finding the shortest path in a graph using optimal substructure; a straight line indicates a single edge; a wavy line indicates a shortest path between the two vertices it connects (among other paths, not shown, sharing the same two vertices); the bold line is the overall shortest path from start to goal.

Overview

Mathematical optimization

In terms of mathematical optimization, dynamic programming usually refers to simplifying a decision by breaking it down into a sequence of decision steps over time.

This is done by defining a sequence of **value functions** V_1, V_2, ..., V_n taking y as an argument representing the **state** of the system at times i from 1 to n.

The definition of $V_n(y)$ is the value obtained in state y at the last time n.

The values V_i at earlier times $i = n - 1, n - 2, \ldots, 2, 1$ can be found by working backwards, using a recursive relationship called the Bellman equation.

For $i = 2, \ldots, n$, V_{i-1} at any state y is calculated from V_i by maximizing a simple function (usually the sum) of the gain from a decision at time $i - 1$ and the function V_i at the new state of the system if this decision is made.

Since V_i has already been calculated for the needed states, the above operation yields V_{i-1} for those states.

Finally, V_1 at the initial state of the system is the value of the optimal solution. The optimal values of the decision variables can be recovered, one by one, by tracking back the calculations already performed.

Control theory

In control theory, a typical problem is to find an admissible control $\mathbf{u}^*$ which causes the system $\dot{\mathbf{x}}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t), t)$ to follow an admissible trajectory $\mathbf{x}^*$ on a continuous time interval $t_0 \leq t \leq t_1$ that minimizes a cost function

$$J = b(\mathbf{x}(t_1), t_1) + \int_{t_0}^{t_1} f(\mathbf{x}(t), \mathbf{u}(t), t)\, \mathrm{d}t$$

The solution to this problem is an optimal control law or policy $\mathbf{u}^* = h(\mathbf{x}(t), t)$, which produces an optimal trajectory $\mathbf{x}^*$ and a cost-to-go function J^*. The latter obeys the fundamental equation of dynamic programming:

$$-J_t^* = \min_{\mathbf{u}} \left\{ f(\mathbf{x}(t), \mathbf{u}(t), t) + J_x^{*\mathsf{T}} \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t), t) \right\}$$

a partial differential equation known as the Hamilton–Jacobi–Bellman equation, in which $J_x^* = \frac{\partial J^*}{\partial \mathbf{x}} = \left[\frac{\partial J^*}{\partial x_1} \quad \frac{\partial J^*}{\partial x_2} \quad \cdots \quad \frac{\partial J^*}{\partial x_n} \right]^{\mathsf{T}}$ and $J_t^* = \frac{\partial J^*}{\partial t}$. One finds that minimizing $\mathbf{u}$ in terms of t, $\mathbf{x}$, and the unknown function J_x^* and then substitutes the result into the Hamilton–Jacobi–Bellman equation to get the partial differential equation to be solved with boundary condition $J(t_1) = b(\mathbf{x}(t_1), t_1)$.[2] In practice, this generally requires numerical techniques for some discrete approximation to the exact optimization relationship.

Alternatively, the continuous process can be approximated by a discrete system, which leads to a following recurrence relation analog to the Hamilton–Jacobi–Bellman equation:

$$J_k^*(\mathbf{x}_{n-k}) = \min_{\mathbf{u}_{n-k}} \left\{ \hat{f}(\mathbf{x}_{n-k}, \mathbf{u}_{n-k}) + J_{k-1}^*(\hat{\mathbf{g}}(\mathbf{x}_{n-k}, \mathbf{u}_{n-k})) \right\}$$

at the k-th stage of n equally spaced discrete time intervals, and where $\hat{f}$ and $\hat{\mathbf{g}}$ denote discrete approximations to f and $\mathbf{g}$. This functional equation is known as the Bellman equation, which can be solved for an exact solution of the discrete approximation of the optimization equation.[3]

Example from economics: Ramsey's problem of optimal saving

In economics, the objective is generally to maximize (rather than minimize) some dynamic social welfare function. In Ramsey's problem, this function relates amounts of consumption to levels of utility. Loosely speaking, the planner faces the trade-off between contemporaneous consumption and future consumption (via investment in capital stock that is used in production), known as intertemporal choice. Future consumption is discounted at a constant rate $\beta \in (0, 1)$. A discrete approximation to the transition equation of capital is given by

$$k_{t+1} = \hat{g}(k_t, c_t) = f(k_t) - c_t$$

where is consumption, k is capital, and f is a production function satisfying the Inada conditions. An initial capital stock is assumed.

Let be consumption in period t, and assume consumption yields utility as long as the consumer lives. Assume the consumer is impatient, so that he discounts future utility by a factor b each period, where . Let be capital in period t. Assume initial capital is a given amount , and suppose that this period's capital and consumption determine next period's capital as , where A is a positive constant and . Assume capital cannot be negative. Then the consumer's decision problem can be written as follows:

subject to for all

Written this way, the problem looks complicated, because it involves solving for all the choice variables . (The capital is not a choice variable—the consumer's initial capital is taken as given.)

The dynamic programming approach to solve this problem involves breaking it apart into a sequence of smaller decisions. To do so, we define a sequence of *value functions* , for which represent the value of having any amount of capital k at each time t. There is (by assumption) no utility from having capital after death, .

The value of any quantity of capital at any previous time can be calculated by backward induction using the Bellman equation. In this problem, for each , the Bellman equation is

subject to

This problem is much simpler than the one we wrote down before, because it involves only two decision variables, and . Intuitively, instead of choosing his whole lifetime plan at birth, the consumer can take things one step at a time. At time *t*, his current capital is given, and he only needs to choose current consumption and saving .

To actually solve this problem, we work backwards. For simplicity, the current level of capital is denoted as *k*. is already known, so using the Bellman equation once we can calculate , and so on until we get to , which is the *value* of the initial decision problem for the whole lifetime. In other words, once we know , we can calculate , which is the maximum of , where is the choice variable and .

Working backwards, it can be shown that the value function at time is

where each is a constant, and the optimal amount to consume at time is

which can be simplified to

We see that it is optimal to consume a larger fraction of current wealth as one gets older, finally consuming all remaining wealth in period *T*, the last period of life.

Computer science

There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub-problems. If a problem can be solved by combining optimal solutions to *non-overlapping* sub-problems, the strategy is called "divide and conquer" instead.[1] This is why merge sort and quick sort are not classified as dynamic programming problems.

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems. Such optimal substructures are usually described by means of recursion. For example, given a graph *G=(V,E)*, the shortest path *p* from a vertex *u* to a vertex *v* exhibits optimal substructure: take any intermediate vertex *w* on this shortest path *p*. If *p* is truly the shortest

path, then it can be split into sub-paths p_1 from u to w and p_2 from w to v such that these, in turn, are indeed the shortest paths between the corresponding vertices (by the simple cut-and-paste argument described in *Introduction to Algorithms*). Hence, one can easily formulate the solution for finding shortest paths in a recursive manner, which is what the Bellman–Ford algorithm or the Floyd–Warshall algorithm does.

Overlapping sub-problems means that the space of sub-problems must be small, that is, any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems. For example, consider the recursive formulation for generating the Fibonacci sequence: $F_i = F_{i-1} + F_{i-2}$, with base case $F_1 = F_2 = 1$. Then $F_{43} = F_{42} + F_{41}$, and $F_{42} = F_{41} + F_{40}$. Now F_{41} is being solved in the recursive sub-trees of both F_{43} as well as F_{42}. Even though the total number of sub-problems is actually small (only 43 of them), we end up solving the same problems over and over if we adopt a naive recursive solution such as this. Dynamic programming takes account of this fact and solves each sub-problem only once.

This can be achieved in either of two ways:

- *Top-down approach*: This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its sub-problems, and if its sub-problems are overlapping, then one can easily memoize or store the solutions to the sub-problems in a table. Whenever we attempt to solve a new sub-problem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the sub-problem and add its solution to the table.
- *Bottom-up approach*: Once we formulate the solution to a problem recursively as in terms of its sub-problems, we can try reformulating the problem in a bottom-up fashion: try solving the sub-problems first and use their solutions to build-on and arrive at solutions to bigger sub-problems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems. For example, if we already know the values of F_{41} and F_{40}, we can directly calculate the value of F_{42}.

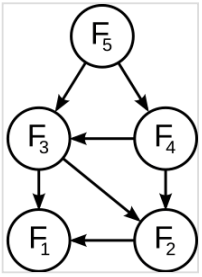


Figure 2. The subproblem graph for the Fibonacci sequence. The fact that it is not a tree indicates overlapping subproblems.

Some programming languages can automatically memoize the result of a function call with a particular set of arguments, in order to speed up call-by-name evaluation (this mechanism is referred to as *call-by-need*). Some languages make it possible portably (e.g. Scheme, Common Lisp, Perl or D). Some languages have automatic memoization built in, such as tabled Prolog and J, which supports memoization with the *M.* adverb.[4] In any case, this is only possible for a referentially transparent function. Memoization is also encountered as an easily accessible design pattern within term-rewrite based languages such as Wolfram Language.

Bioinformatics

Dynamic programming is widely used in bioinformatics for tasks such as sequence alignment, protein folding, RNA structure prediction and protein-DNA binding. The first dynamic programming algorithms for protein-DNA binding were developed in the 1970s independently by Charles DeLisi in US[5] and Georgii Gurskii and Alexander Zasedatelev in USSR.[6] Recently these algorithms have become very popular in bioinformatics and computational biology, particularly in the studies of nucleosome positioning and transcription factor binding.

Examples: computer algorithms

Dijkstra's algorithm for the shortest path problem

From a dynamic programming point of view, Dijkstra's algorithm for the shortest path problem is a successive approximation scheme that solves the dynamic programming functional equation for the shortest path problem by the **Reaching** method.[7][8][9]

In fact, Dijkstra's explanation of the logic behind the algorithm,[10] namely

Problem 2. Find the path of minimum total length between two given nodes and .

> We use the fact that, if is a node on the minimal path from to , knowledge of the latter implies the knowledge of the minimal path from to .

is a paraphrasing of Bellman's famous Principle of Optimality in the context of the shortest path problem.

Fibonacci sequence

Using dynamic programming in the calculation of the *n*th member of the Fibonacci sequence improves its performance greatly. Here is a naïve implementation, based directly on the mathematical definition:

```
function fib(n)
    if n <= 1 return n
    return fib(n − 1) + fib(n − 2)
```

Notice that if we call, say, `fib(5)`, we produce a call tree that calls the function on the same value many different times:

1. `fib(5)`
2. `fib(4) + fib(3)`
3. `(fib(3) + fib(2)) + (fib(2) + fib(1))`
4. `((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))`
5. `(((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))`

In particular, `fib(2)` was calculated three times from scratch. In larger examples, many more values of `fib`, or *subproblems*, are recalculated, leading to an exponential time algorithm.

Now, suppose we have a simple map object, *m*, which maps each value of `fib` that has already been calculated to its result, and we modify our function to use it and update it. The resulting function requires only O(*n*) time instead of exponential time (but requires O(*n*) space):

```
var m := map(0 → 0, 1 → 1)
function fib(n)
    if key n is not in map m
        m[n] := fib(n − 1) + fib(n − 2)
    return m[n]
```

This technique of saving values that have already been calculated is called *memoization*; this is the top-down approach, since we first break the problem into subproblems and then calculate and store values.

In the **bottom-up** approach, we calculate the smaller values of `fib` first, then build larger values from them. This method also uses O(*n*) time since it contains a loop that repeats `n − 1` times, but it only takes constant (O(1)) space, in contrast to the top-down approach which requires O(*n*) space to store the map.

```
function fib(n)
    if n = 0
        return 0
    else
        var previousFib := 0, currentFib := 1
        repeat n − 1 times // loop is skipped if n = 1
            var newFib := previousFib + currentFib
            previousFib := currentFib
            currentFib  := newFib
        return currentFib
```

In both examples, we only calculate `fib(2)` one time, and then use it to calculate both `fib(4)` and `fib(3)`, instead of computing it every time either of them is evaluated.

A type of balanced 0–1 matrix

Consider the problem of assigning values, either zero or one, to the positions of an $n \times n$ matrix, with n even, so that each row and each column contains exactly $n / 2$ zeros and $n / 2$ ones. We ask how many different assignments there are for a given n. For example, when $n = 4$, five possible solutions are

There are at least three possible approaches: brute force, backtracking, and dynamic programming.

Brute force consists of checking all assignments of zeros and ones and counting those that have balanced rows and columns ($n / 2$ zeros and $n / 2$ ones). As there are possible assignments and sensible assignments, this strategy is not practical except maybe up to .

Backtracking for this problem consists of choosing some order of the matrix elements and recursively placing ones or zeros, while checking that in every row and column the number of elements that have not been assigned plus the number of ones or zeros are both at least $n / 2$. While more sophisticated than brute force, this approach will visit every solution once, making it impractical for n larger than six, since the number of solutions is already 116,963,796,250 for $n = 8$, as we shall see.

Dynamic programming makes it possible to count the number of solutions without visiting them all. Imagine backtracking values for the first row – what information would we require about the remaining rows, in order to be able to accurately count the solutions obtained for each first row value? We consider $k \times n$ boards, where $1 \leq k \leq n$, whose k rows contain zeros and ones. The function f to which memoization is applied maps vectors of n pairs of integers to the number of admissible boards (solutions). There is one pair for each column, and its two components indicate respectively the number of zeros and ones that have yet to be placed in that column. We seek the value of (n arguments or one vector of n elements). The process of subproblem creation involves iterating over every one of possible assignments for the top row of the board, and going through every column, subtracting one from the appropriate element of the pair for that column, depending on whether the assignment for the top row contained a zero or a one at that position. If any one of the results is negative, then the assignment is invalid and does not contribute to the set of solutions (recursion stops). Otherwise, we have an assignment for the top row of the $k \times n$ board and recursively compute the number of solutions to the remaining $(k - 1) \times n$ board, adding the numbers of solutions for every admissible assignment of the top row and returning the sum, which is being memoized. The base case is the trivial subproblem, which occurs for a $1 \times n$ board. The number of solutions for this board is either zero or one, depending on whether the vector is a permutation of $n / 2$ and $n / 2$ pairs or not.

For example, in the first two boards shown above the sequences of vectors would be

```
((2, 2) (2, 2) (2, 2) (2, 2))       ((2, 2) (2, 2) (2, 2) (2, 2))     k = 4
  0      1      0      1              0      0      1      1

((1, 2) (2, 1) (1, 2) (2, 1))       ((1, 2) (1, 2) (2, 1) (2, 1))     k = 3
  1      0      1      0              0      0      1      1

((1, 1) (1, 1) (1, 1) (1, 1))       ((0, 2) (0, 2) (2, 0) (2, 0))     k = 2
  0      1      0      1              1      1      0      0

((0, 1) (1, 0) (0, 1) (1, 0))       ((0, 1) (0, 1) (1, 0) (1, 0))     k = 1
  1      0      1      0              1      1      0      0

((0, 0) (0, 0) (0, 0) (0, 0))       ((0, 0) (0, 0), (0, 0) (0, 0))
```

The number of solutions (sequence A058527 in the OEIS) is

Links to the MAPLE implementation of the dynamic programming approach may be found among the external links.

Checkerboard

Consider a checkerboard with $n \times n$ squares and a cost function `c(i, j)` which returns a cost associated with square `(i,j)` (*i* being the row, *j* being the column). For instance (on a 5×5 checkerboard),

5	6	7	4	7	8
4	7	6	1	1	4
3	3	5	7	8	2
2	–	6	7	0	–
1	–	–	**5**	–	–
	1	**2**	**3**	**4**	**5**

Thus `c(1, 3) = 5`

Let us say there was a checker that could start at any square on the first rank (i.e., row) and you wanted to know the shortest path (the sum of the minimum costs at each visited rank) to get to the last rank; assuming the checker could move only diagonally left forward, diagonally right forward, or straight forward. That is, a checker on `(1,3)` can move to `(2,2)`, `(2,3)` or `(2,4)`.

5					
4					
3					
2		x	x	x	
1			o		
	1	**2**	**3**	**4**	**5**

This problem exhibits **optimal substructure**. That is, the solution to the entire problem relies on solutions to subproblems. Let us define a function `q(i, j)` as

> *q*(*i*, *j*) = the minimum cost to reach square (*i*, *j*).

Starting at rank `n` and descending to rank `1`, we compute the value of this function for all the squares at each successive rank. Picking the square that holds the minimum value at each rank gives us the shortest path between rank `n` and rank `1`.

The function `q(i, j)` is equal to the minimum cost to get to any of the three squares below it (since those are the only squares that can reach it) plus `c(i, j)`. For instance:

5					
4			A		
3		B	C	D	
2					
1					
	1	**2**	**3**	**4**	**5**

Now, let us define `q(i, j)` in somewhat more general terms:

The first line of this equation deals with a board modeled as squares indexed on `1` at the lowest bound and `n` at the highest bound. The second line specifies what happens at the first rank; providing a base case. The third line, the recursion, is the important part. It represents the `A,B,C,D` terms in the example. From this definition we can derive straightforward recursive code for `q(i, j)`. In the following pseudocode, `n` is the size of the board, `c(i, j)` is the cost function, and `min()` returns the minimum of a number of values:

```
function minCost(i, j)
    if j < 1 or j > n
        return infinity
    else if i = 1
        return c(i, j)
    else
        return min( minCost(i-1, j-1), minCost(i-1, j), minCost(i-1, j+1) ) + c(i, j)
```

This function only computes the path cost, not the actual path. We discuss the actual path below. This, like the Fibonacci-numbers example, is horribly slow because it too exhibits the **overlapping sub-problems** attribute. That is, it recomputes the same path costs over and over. However, we can compute it much faster in a bottom-up fashion if we store path costs in a two-dimensional array `q[i, j]` rather than using a function. This avoids recomputation; all the values needed for array `q[i, j]` are computed ahead of time only once. Precomputed values for `(i,j)` are simply looked up whenever needed.

We also need to know what the actual shortest path is. To do this, we use another array `p[i, j]`; a *predecessor array*. This array records the path to any square `s`. The predecessor of `s` is modeled as an offset relative to the index (in `q[i, j]`) of the precomputed path cost of `s`. To reconstruct the complete path, we lookup the predecessor of `s`, then the predecessor of that square, then the predecessor of that square, and so on recursively, until we reach the starting square. Consider the following pseudocode:

```
function computeShortestPathArrays()
    for x from 1 to n
        q[1, x] := c(1, x)
    for y from 1 to n
        q[y, 0]     := infinity
        q[y, n + 1] := infinity
    for y from 2 to n
        for x from 1 to n
            m := min(q[y-1, x-1], q[y-1, x], q[y-1, x+1])
            q[y, x] := m + c(y, x)
            if m = q[y-1, x-1]
                p[y, x] := -1
            else if m = q[y-1, x]
                p[y, x] :=  0
            else
                p[y, x] :=  1
```

Now the rest is a simple matter of finding the minimum and printing it.

```
function computeShortestPath()
    computeShortestPathArrays()
    minIndex := 1
    min := q[n, 1]
    for i from 2 to n
        if q[n, i] < min
            minIndex := i
            min := q[n, i]
    printPath(n, minIndex)
```

```
function printPath(y, x)
    print(x)
    print("<-")
    if y = 2
        print(x + p[y, x])
```

```
    else
        printPath(y-1, x + p[y, x])
```

Sequence alignment

In genetics, sequence alignment is an important application where dynamic programming is essential.[11] Typically, the problem consists of transforming one sequence into another using edit operations that replace, insert, or remove an element. Each operation has an associated cost, and the goal is to find the sequence of edits with the lowest total cost.

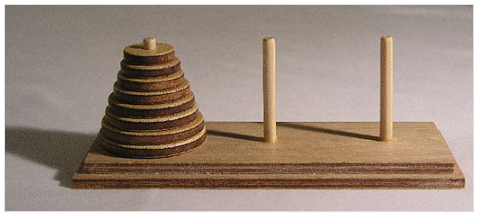
The problem can be stated naturally as a recursion, a sequence A is optimally edited into a sequence B by either:

1. inserting the first character of B, and performing an optimal alignment of A and the tail of B
2. deleting the first character of A, and performing the optimal alignment of the tail of A and B
3. replacing the first character of A with the first character of B, and performing optimal alignments of the tails of A and B.

The partial alignments can be tabulated in a matrix, where cell (i,j) contains the cost of the optimal alignment of A[1..i] to B[1..j]. The cost in cell (i,j) can be calculated by adding the cost of the relevant operations to the cost of its neighboring cells, and selecting the optimum.

Different variants exist, see Smith–Waterman algorithm and Needleman–Wunsch algorithm.

Tower of Hanoi puzzle



A model set of the Towers of Hanoi (with 8 disks)

The **Tower of Hanoi** or **Towers of Hanoi** is a mathematical game or puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

An animated solution of the **Tower of Hanoi** puzzle for *T(4,3)*.

The dynamic programming solution consists of solving the functional equation

S(n,h,t) = S(n-1,h, not(h,t)) ; S(1,h,t) ; S(n-1,not(h,t),t)

where n denotes the number of disks to be moved, h denotes the home rod, t denotes the target rod, not(h,t) denotes the third rod (neither h nor t), ";" denotes concatenation, and

S(n, h, t) := solution to a problem consisting of n disks that are to be moved from rod h to rod t.

For n=1 the problem is trivial, namely S(1,h,t) = "move a disk from rod h to rod t" (there is only one disk left).

The number of moves required by this solution is $2^n - 1$. If the objective is to **maximize** the number of moves (without cycling) then the dynamic programming functional equation is slightly more complicated and $3^n - 1$ moves are required.[12]

Egg dropping puzzle

The following is a description of the instance of this famous puzzle involving N=2 eggs and a building with H=36 floors:[13]

> Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing (using U.S. English terminology, in which the first floor is at ground level). We make a few assumptions:
>
> - An egg that survives a fall can be used again.
> - A broken egg must be discarded.
> - The effect of a fall is the same for all eggs.
> - If an egg breaks when dropped, then it would break if dropped from a higher window.
> - If an egg survives a fall, then it would survive a shorter fall.
> - It is not ruled out that the first-floor windows break eggs, nor is it ruled out that eggs can survive the 36th-floor windows.
>
> If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second-floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the lowest number of egg-droppings that is guaranteed to work in all cases?

To derive a dynamic programming functional equation for this puzzle, let the **state** of the dynamic programming model be a pair s = (n,k), where

> n = number of test eggs available, $n = 0, 1, 2, 3, \ldots, N - 1$.
> k = number of (consecutive) floors yet to be tested, $k = 0, 1, 2, \ldots, H - 1$.

For instance, $s = (2,6)$ indicates that two test eggs are available and 6 (consecutive) floors are yet to be tested. The initial state of the process is $s = (N,H)$ where N denotes the number of test eggs available at the commencement of the experiment. The process terminates either when there are no more test eggs ($n = 0$) or when $k = 0$, whichever occurs first. If termination occurs at state $s = (0,k)$ and $k > 0$, then the test failed.

Now, let

> $W(n,k)$ = minimum number of trials required to identify the value of the critical floor under the worst-case scenario given that the process is in state $s = (n,k)$.

Then it can be shown that[14]

$$W(n,k) = 1 + \min\{\max(W(n-1, x-1), W(n,k-x)): x = 1, 2, \ldots, k\}$$

with $W(n,0) = 0$ for all $n > 0$ and $W(1,k) = k$ for all k. It is easy to solve this equation iteratively by systematically increasing the values of n and k.

Faster DP solution using a different parametrization

Notice that the above solution takes time with a DP solution. This can be improved to time by binary searching on the optimal in the above recurrence, since is increasing in while is decreasing in , thus a local minimum of is a global minimum. Also, by storing the optimal for each cell in the DP table and referring to its value for the previous cell, the optimal for each cell can be found in constant time, improving it to time. However, there is an even faster solution that involves a different parametrization of the problem:

Let k be the total number of floors such that the eggs break when dropped from the kth floor (The example above is equivalent to taking).

Let be the minimum floor from which the egg must be dropped to be broken.

Let be the maximum number of values of that are distinguishable using t tries and n eggs.

Then for all .

Let be the floor from which the first egg is dropped in the optimal strategy.

If the first egg broke, is from to and distinguishable using at most tries and eggs.

If the first egg did not break, is from to k and distinguishable using tries and n eggs.

Therefore, .

Then the problem is equivalent to finding the minimum such that .

To do so, we could compute in order of increasing t, which would take time.

Thus, if we separately handle the case of , the algorithm would take time.

But the recurrence relation can in fact be solved, giving , which can be computed in

time using the identity for all .

Since for all , we can binary search on t to find , giving an algorithm.[15]

Matrix chain multiplication

Matrix chain multiplication is a well-known example that demonstrates utility of dynamic programming. For example, engineering applications often have to multiply a chain of matrices. It is not surprising to find matrices of large dimensions, for example 100×100. Therefore, our task is to multiply matrices . Matrix multiplication is not commutative, but is associative; and we can multiply only two matrices at a time. So, we can multiply this chain of matrices in many different ways, for example:

$((A_1 \times A_2) \times A_3) \times \dots A_n$

$A_1 \times (((A_2 \times A_3) \times \dots) \times A_n)$

$(A_1 \times A_2) \times (A_3 \times \dots A_n)$

and so on. There are numerous ways to multiply this chain of matrices. They will all produce the same final result, however they will take more or less time to compute, based on which particular matrices are multiplied. If matrix A has dimensions m×n and matrix B has dimensions n×q, then matrix C=A×B will have dimensions m×q, and will require m*n*q scalar multiplications (using a simplistic matrix multiplication algorithm for purposes of illustration).

For example, let us multiply matrices A, B and C. Let us assume that their dimensions are m×n, n×p, and p×s, respectively. Matrix A×B×C will be of size m×s and can be calculated in two ways shown below:

1. Ax(B×C) This order of matrix multiplication will require nps + mns scalar multiplications.
2. (A×B)×C This order of matrix multiplication will require mnp + mps scalar calculations.

Let us assume that m = 10, n = 100, p = 10 and s = 1000. So, the first way to multiply the chain will require 1,000,000 + 1,000,000 calculations. The second way will require only 10,000+100,000 calculations. Obviously, the second way is faster, and we should multiply the matrices using that arrangement of parenthesis.

Therefore, our conclusion is that the order of parenthesis matters, and that our task is to find the optimal order of parenthesis.

At this point, we have several choices, one of which is to design a dynamic programming algorithm that will split the problem into overlapping problems and calculate the optimal arrangement of parenthesis. The dynamic programming solution is presented below.

Let's call m[i,j] the minimum number of scalar multiplications needed to multiply a chain of matrices from matrix i to matrix j (i.e. $A_i \times .... \times A_j$, i.e. i<=j). We split the chain at some matrix k, such that i <= k < j, and try to find out which combination produces minimum m[i,j].

The formula is:

```
      if i = j, m[i,j]= 0
      if i < j, m[i,j]= min over all possible values of k (m[i,k]+m[k+1,j] +           )
```

where *k* ranges from *i* to $j - 1$.

- is the row dimension of matrix i,
- is the column dimension of matrix k,
- is the column dimension of matrix j.

This formula can be coded as shown below, where input parameter "chain" is the chain of matrices, i.e. :

```
function OptimalMatrixChainParenthesis(chain)
    n = length(chain)
    for i = 1, n
        m[i,i] = 0    // Since it takes no calculations to multiply one matrix
    for len = 2, n
        for i = 1, n − len + 1
            j = i + len −1
            m[i,j] = infinity      // So that the first calculation updates
            for k = i, j−1
                q = m[i, k] + m[k+1, j] +
                if q < m[i, j]     // The new order of parentheses is better than what we had
                    m[i, j] = q    // Update
                    s[i, j] = k    // Record which k to split on, i.e. where to place the parenthesis
```

So far, we have calculated values for all possible $m[i, j]$, the minimum number of calculations to multiply a chain from matrix *i* to matrix *j*, and we have recorded the corresponding "split point"$s[i, j]$. For example, if we are multiplying chain $A_1 \times A_2 \times A_3 \times A_4$, and it turns out that $m[1, 3] = 100$ and $s[1, 3] = 2$, that means that the optimal placement of parenthesis for matrices 1 to 3 is and to multiply those matrices will require 100 scalar calculations.

This algorithm will produce "tables" *m*[,] and *s*[,] that will have entries for all possible values of i and j. The final solution for the entire chain is m[1, n], with corresponding split at s[1, n]. Unraveling the solution will be recursive, starting from the top and continuing until we reach the base case, i.e. multiplication of single matrices.

Therefore, the next step is to actually split the chain, i.e. to place the parenthesis where they (optimally) belong. For this purpose we could use the following algorithm:

```
function PrintOptimalParenthesis(s, i, j)
    if i = j
        print "A"i
    else
        print "("
        PrintOptimalParenthesis(s, i, s[i, j])
        PrintOptimalParenthesis(s, s[i, j] + 1, j)
        print ")"
```

Of course, this algorithm is not useful for actual multiplication. This algorithm is just a user-friendly way to see what the result looks like.

To actually multiply the matrices using the proper splits, we need the following algorithm:

```
    function MatrixChainMultiply(chain from 1 to n)       // returns the final matrix, i.e. A1×A2×... ×An
       OptimalMatrixChainParenthesis(chain from 1 to n)   // this will produce s[ . ] and m[ . ] "tables"
       OptimalMatrixMultiplication(s, chain from 1 to n)  // actually multiply

    function OptimalMatrixMultiplication(s, i, j)   // returns the result of multiplying a chain of matrices from Ai to
Aj in optimal way
       if i < j
          // keep on splitting the chain and multiplying the matrices in left and right sides
          LeftSide = OptimalMatrixMultiplication(s, i, s[i, j])
          RightSide = OptimalMatrixMultiplication(s, s[i, j] + 1, j)
          return MatrixMultiply(LeftSide, RightSide)
       else if i = j
          return Ai   // matrix at position i
       else
          print "error, i <= j must hold"

     function MatrixMultiply(A, B)    // function that multiplies two matrices
       if columns(A) = rows(B)
          for i = 1, rows(A)
             for j = 1, columns(B)
                C[i, j] = 0
                for k = 1, columns(A)
                    C[i, j] = C[i, j] + A[i, k]*B[k, j]
                return C
       else
           print "error, incompatible dimensions."
```

History of the name

The term *dynamic programming* was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another. By 1953, he refined this to the modern meaning, referring specifically to nesting smaller decision problems inside larger decisions,[16] and the field was thereafter recognized by the IEEE as a systems analysis and engineering topic. Bellman's contribution is remembered in the name of the Bellman equation, a central result of dynamic programming which restates an optimization problem in recursive form.

Bellman explains the reasoning behind the term *dynamic programming* in his autobiography, *Eye of the Hurricane: An Autobiography*:

> I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, "Where did the name, dynamic programming, come from?" The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word "research". I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.
>
> —Richard Bellman, *Eye of the Hurricane: An Autobiography* (1984, page 159)

The word *dynamic* was chosen by Bellman to capture the time-varying aspect of the problems, and because it sounded impressive.[11] The word *programming* referred to the use of the method to find an optimal *program*, in the sense of a military schedule for training or logistics. This usage is the same as that in the phrases *linear programming* and *mathematical programming*, a synonym for mathematical optimization.[17]

The above explanation of the origin of the term may be inaccurate: According to Russell and Norvig, the above story "cannot be strictly true, because his first paper using the term (Bellman, 1952) appeared before Wilson became Secretary of Defense in 1953."[18] Also, Harold J. Kushner (http://a2c2.org/awards/richard-e-bellman-control-heritage-award/2004-00-00t000000/harold-j-kushner) Archived (https://web.archive.org/web/20170301091951/http://a2c2.org/awards/richard-e-bellman-control-heritage-award/2004-00-00t000000/harold-j-kushner) 2017-03-01 at the Wayback Machine stated in a speech that, "On the other hand, when I asked [Bellman] the same question, he replied that he was trying to upstage Dantzig's linear programming by adding dynamic. Perhaps both motivations were true."

See also

Systems science portal

Mathematics portal

- Convexity in economics – Significant topic in economics
- Greedy algorithm – Sequence of locally optimal choices
- Non-convexity (economics) – Violations of the convexity assumptions of elementary economics
- Stochastic programming – Framework for modeling optimization problems that involve uncertainty
- Stochastic dynamic programming – 1957 technique for modelling problems of decision making under uncertainty
- Reinforcement learning – Field of machine learning

References

1. Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. (2001), Introduction to Algorithms (2nd ed.), MIT Press & McGraw–Hill, ISBN 0-262-03293-7 . pp. 344.
2. Kamien, M. I.; Schwartz, N. L. (1991). *Dynamic Optimization: The Calculus of Variations and Optimal Control in Economics and Management* (https://books.google.com/books?id=0IoGUn8wjDQC&pg=PA261) (Second ed.). New York: Elsevier. p. 261. ISBN 978-0-444-01609-6.
3. Kirk, Donald E. (1970). *Optimal Control Theory: An Introduction* (https://books.google.com/books?id=fCh2SAtWIdwC&pg=PA94). Englewood Cliffs, NJ: Prentice-Hall. pp. 94–95. ISBN 978-0-13-638098-6.
4. "M. Memo" (http://www.jsoftware.com/help/dictionary/dmcapdot.htm). *J Vocabulary*. J Software. Retrieved 28 October 2011.
5. DeLisi, Biopolymers, 1974, Volume 13, Issue 7, pages 1511–1512, July 1974
6. Gurskiĭ GV, Zasedatelev AS, Biofizika, 1978 Sep-Oct;23(5):932-46
7. Sniedovich, M. (2006), "Dijkstra's algorithm revisited: the dynamic programming connexion" (http://matwbn.icm.edu.pl/ksiazki/cc/cc35/cc3536.pdf) (PDF), *Journal of Control and Cybernetics*, **35** (3): 599–620. Online version of the paper with interactive computational modules. (http://www.ifors.ms.unimelb.edu.au/tutorial/dijkstra_new/index.html)
8. Denardo, E.V. (2003), *Dynamic Programming: Models and Applications*, Mineola, NY: Dover Publications, ISBN 978-0-486-42810-9
9. Sniedovich, M. (2010), *Dynamic Programming: Foundations and Principles*, Taylor & Francis, ISBN 978-0-8247-4099-3
10. Dijkstra, E. W. (December 1959). "A note on two problems in connexion with graphs". *Numerische Mathematik*. **1** (1): 269–271. doi:10.1007/BF01386390 (https://doi.org/10.1007%2FBF01386390).
11. Eddy, S. R. (2004). "What is Dynamic Programming?". *Nature Biotechnology*. **22** (7): 909–910. doi:10.1038/nbt0704-909 (https://doi.org/10.1038%2Fnbt0704-909). PMID 15229554 (https://pubmed.ncbi.nlm.nih.gov/15229554). S2CID 5352062 (https://api.semanticscholar.org/CorpusID:5352062).
12. Moshe Sniedovich (2002), "OR/MS Games: 2. The Towers of Hanoi Problem", *INFORMS Transactions on Education*, **3** (1): 34–51, doi:10.1287/ited.3.1.45 (https://doi.org/10.1287%2Fited.3.1.45).
13. Konhauser J.D.E., Velleman, D., and Wagon, S. (1996). Which way did the Bicycle Go? (https://books.google.com/books?id=ElSi5V5uS2MC) Dolciani Mathematical Expositions – No 18. The Mathematical Association of America.

14. Sniedovich, Moshe (2003). "OR/MS Games: 4. The Joy of Egg-Dropping in Braunschweig and Hong Kong" (https://doi.org/10.1287%2Fited.4.1.48). *INFORMS Transactions on Education*. **4**: 48–64. doi:10.1287/ited.4.1.48 (https://doi.org/10.1287%2Fited.4.1.48).
15. Dean Connable Wills, *Connections between combinatorics of permutations and algorithms and geometry* (https://ir.library.oregonstate.edu/xmlui/handle/1957/11929?show=full)
16. Stuart Dreyfus. "Richard Bellman on the birth of Dynamical Programming" (https://web.archive.org/web/20050110161049/http://www.wu-wien.ac.at/usr/h99c/h9951826/bellman_dynprog.pdf).
17. Nocedal, J.; Wright, S. J. (2006). *Numerical Optimization* (https://archive.org/details/numericaloptimiz00noce_639). Springer. p. 9 (https://archive.org/details/numericaloptimiz00noce_639/page/n21). ISBN 9780387303031.
18. Russell, S.; Norvig, P. (2009). *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall. ISBN 978-0-13-207148-2.

Further reading

- Adda, Jerome; Cooper, Russell (2003), *Dynamic Economics* (https://mitpress.mit.edu/books/dynamic-economics), MIT Press, ISBN 9780262012010. An accessible introduction to dynamic programming in economics. MATLAB code for the book (https://sites.google.com/site/coopereconomics/matlab-programs) Archived (https://web.archive.org/web/20201009085820/https://sites.google.com/site/coopereconomics/matlab-programs) 2020-10-09 at the Wayback Machine.
- Bellman, Richard (1954), "The theory of dynamic programming", *Bulletin of the American Mathematical Society*, **60** (6): 503–516, doi:10.1090/S0002-9904-1954-09848-8 (https://doi.org/10.1090%2FS0002-9904-1954-09848-8), MR 0067459 (https://mathscinet.ams.org/mathscinet-getitem?mr=0067459). Includes an extensive bibliography of the literature in the area, up to the year 1954.
- Bellman, Richard (1957), *Dynamic Programming*, Princeton University Press. Dover paperback edition (2003), ISBN 0-486-42809-5.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), *Introduction to Algorithms* (2nd ed.), MIT Press & McGraw–Hill, ISBN 978-0-262-03293-3. Especially pp. 323–69.
- Dreyfus, Stuart E.; Law, Averill M. (1977), *The Art and Theory of Dynamic Programming*, Academic Press, ISBN 978-0-12-221860-6.
- Giegerich, R.; Meyer, C.; Steffen, P. (2004), "A Discipline of Dynamic Programming over Sequence Data" (http://bibiserv.techfak.uni-bielefeld.de/adp/ps/GIE-MEY-STE-2004.pdf) (PDF), *Science of Computer Programming*, **51** (3): 215–263, doi:10.1016/j.scico.2003.12.005 (https://doi.org/10.1016%2Fj.scico.2003.12.005).
- Meyn, Sean (2007), *Control Techniques for Complex Networks* (https://web.archive.org/web/20100619011046/https://netfiles.uiuc.edu/meyn/www/spm_files/CTCN/CTCN.html), Cambridge University Press, ISBN 978-0-521-88441-9, archived from the original (https://netfiles.uiuc.edu/meyn/www/spm_files/CTCN/CTCN.html) on 2010-06-19.
- Sritharan, S. S. (1991). "Dynamic Programming of the Navier-Stokes Equations". *Systems and Control Letters*. **16** (4): 299–307. doi:10.1016/0167-6911(91)90020-f (https://doi.org/10.1016%2F0167-6911%2891%2990020-f).
- Stokey, Nancy; Lucas, Robert E.; Prescott, Edward (1989), *Recursive Methods in Economic Dynamics*, Harvard Univ. Press, ISBN 978-0-674-75096-8.

External links

- A Tutorial on Dynamic programming (http://mat.gsia.cmu.edu/classes/dynamic/dynamic.html)
- MIT course on algorithms (https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/resources/lecture-15-dynamic-programming-part-1-srtbot-fib-dags-bowling/) - Includes 4 video lectures on DP, lectures 15–18
- Applied Mathematical Programming (http://web.mit.edu/15.053/www/AMP.htm) by Bradley, Hax, and Magnanti, Chapter 11 (http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf)
- More DP Notes (http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamic)
- King, Ian, 2002 (1987), "A Simple Introduction to Dynamic Programming in Macroeconomic Models. (https://researchspace.auckland.ac.nz/bitstream/handle/2292/190/230.pdf)" An introduction to dynamic programming as an important tool in economic theory.
- Dynamic Programming: from novice to advanced (http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=dynProg) A TopCoder.com article by Dumitru on Dynamic Programming

- Algebraic Dynamic Programming (https://bibiserv.cebitec.uni-bielefeld.de/adp/welcome.html) – a formalized framework for dynamic programming, including an entry-level course (https://bibiserv.cebitec.uni-bielefeld.de/cgi-bin/dpcourse) to DP, University of Bielefeld
- Dreyfus, Stuart, "Richard Bellman on the birth of Dynamic Programming. (http://www.cas.mcmaster.ca/~se3c03/journal_papers/dy_birth.pdf) Archived (https://web.archive.org/web/20201013233916/http://www.cas.mcmaster.ca/~se3c03/journal_papers/dy_birth.pdf) 2020-10-13 at the Wayback Machine"
- Dynamic programming tutorial (https://web.archive.org/web/20080626183359/http://www.avatar.se/lectures/molbioinfo2001/dynprog/dynamic.html)
- A Gentle Introduction to Dynamic Programming and the Viterbi Algorithm (http://www.cambridge.org/resources/0521882672/7934_kaeslin_dynpro_new.pdf)
- Tabled Prolog BProlog (http://www.probp.com), XSB (http://xsb.sourceforge.net/), SWI-Prolog (https://www.swi-prolog.org/pldoc/man?section=tabling)
- IFORS online interactive dynamic programming modules (https://ifors.ms.unimelb.edu.au/tutorial/) including, shortest path, traveling salesman, knapsack, false coin, egg dropping, bridge and torch, replacement, chained matrix products, and critical path problem.
