julia> Lecture Seven: Dynamic Programming

Course: Computational Bootcamp
Name: John Higgins
Date: July 8, 2024

julia> Today's goals

1. Understand the basics of dynamic programming
2. Understand how it can be utilized to solve complex economic problems

julia> Dynamic Programming Intro

- Optimization problems in our economic models are often complicated
  - Many state variables
  - Many possible choices at each state
  - Future decisions depend on previous decisions
  - Agents are forward looking and care about future pay-offs
- Large number of calculations are necessary to find optimal decision rules
- We want to solve these problems as quickly as possible

julia> Dynamic Programming

- Dynamic programming simplifies the problem by:
  1. Breaking it down into smaller sub-problems
  2. Solving these sub-problems recursively
- Example: Suppose you want to make saving decision from age 25 to 65:
- You could search over all possible savings paths.
- OR...
  1. First solve for optimal savings rule at age 65
  2. Then given age 65 savings rules, solve for age 64 savings rule...
  3. Giving all future savings rules, solve for age 25 savings rule

julia> Dynamic Programming

- The general idea:
  1. Break down the problem into smaller sub-problems
  2. Start by solving the easiest of the sub-problems
  3. Store results, which you will use to solve more sub-problems
- This is often much faster than just searching over all possible decisions
- Many problems would be computationally infeasible without dynamic programming
- Today we will go through some classic dynamic programming problems

julia> Fibonacci Sequence

> Each number is the sum of the previous two numbers:

$$F(n) = F(n-1) + F(n-2)$$

> The Fibonacci sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

> To compute this, we could do it via brute force for each n

> However, saving the result of the previous sub-problem can dramatically speed things up

> Key takeaway: if a problem is recursive, make use of that!

julia> Egg Dropping Problem

- We have n eggs and there are k floors in a building (as one does)
- We want to know the highest possible from which we can drop the egg without it breaking
- Want to do this as quickly as possible, so minimize the number of times we drop an egg
- We also never want to risk running out of eggs
- Rules:
  - Eggs that survive can be used again, but broken eggs are discarded
  - If an egg breaks on a floor, it would break on all higher floors
  - If an egg doesn't break on a floor, it wouldn't break on all lower floors

Let $E(n, k)$ be the answer to the problem julia> Solution

- Suppose we have 1 egg and k floors.
  - We have to drop from every floor from the bottom (remember, we can't risk running out of eggs)

$$E(1,k) = k$$

  - Suppose we have 2 eggs and k floors
  - If we drop at floor j, either:
    1. The egg breaks. We have 1 egg for $j-1$ floors
    2. The egg survives. We have 2 eggs to search $k-j$ floors

$$E(2,k) = \min_j\{\max E(1,j-1), E(2,k-j)\}$$

  - General formula is then:

$$E(n,k) = \min_j\{\max\{E(n-1,j-1), E(n,k-j)\}\}$$

julia> Shortest Travel Time

- Suppose we are interested in the shortest travel time between a source city C_1 and and some other cities $\{C_j\}$
- Direct route is not always the fastest because of traffic and terrain
- The direct road distance between cities x, y is given by $d(x, y)$
- If direct road doesn't exist, say $d(x, y) = \infty$
- We could check all possible routes from C_1 to C_j for all j, but that would take a while

julia> Dijkstra's Algorithm

> Start at source city C_1 and calculate direct distance to all cities:

$$M(1, j) = d(C_1, C_j)$$

> Second, visit the city with shortest distance to C_1 that is not the source city. Call it C_v

$$C_v = \arg\min_j M(1, j)$$

> Update distance for all cities j if it is faster to go through C_v

$$M(2, j) = \min\{M(1, j), M(1, v) + d(v, j)\}$$

julia> Dijkstra's Algorithm, continued

- Next, go to the unvisited city with shortest distance to C_1 that is not the source city or a previously visited city

$$C_v = \arg\min_j M(2, j)$$

- Again, update distances if it is faster to travel first through C_v. If not, go the previously-found fastest way
- Repeat until we arrive at the minimum travel distance to get to all the cities!

$$M(N, j) = \min\{M(N-1, j), M(N-1, v) + d(v, j)\}$$