Dynamic Programming

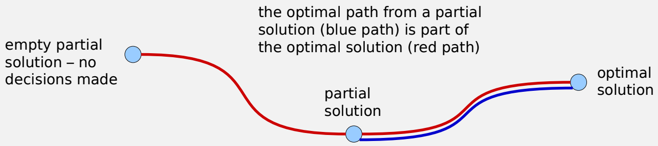
We combine several elements –

- an optimization problem
- the need for exhaustive search (greedy is insufficient)
- the existence of repeated subproblems – it is possible to arrive at a given subproblem through different series of choices
  - what matters for solving a subproblem is the state resulting from the partial solution, not the partial solution itself
- memoization – so each subproblem only needs to be solved once

---

Optimal Substructure

- any series-of-choices formulation requires *optimal substructure* – an optimal solution can be constructed from optimal solutions of subproblems



Dynamic Programming vs Recursive Backtracking

Both begin with the same recursive formulation –

- solution is constructed by making a series of decisions
- you consider the next possibilities for the current decision, then ask friends to solve the problem given the consequences of each choice

The difference is how the subproblems are parameterized and enumerated –

- recursive backtracking uses a depth-first search of the solution space
  - subproblems depend on the series of decisions made
  - may end up enumerating all possible series of decisions
- dynamic programming iterates through the states
  - subproblems depend on the state resulting from the series of decisions
  - enumerates all possible states

16(ish) Steps to Dynamic Programming Success

establishing the problem

1. specifications
   - input
   - output
   - legal solution
   - optimization goal
2. examples
3. size

brainstorming ideas

4. targets
5. tactics
6. approaches

defining the algorithm

7. generalize / define subproblems
   - partial solution
   - alternatives
   - subproblem
8. base case(s)
9. main case
10. top level
    - initial subproblem
    - setup
    - wrapup
11. special cases
12. ~~*algorithm*~~

showing correctness

13. termination
    - making progress
    - reaching the end
14. correctness
    - establish the base case(s)
    - show the main case
    - final answer

determining efficiency

15. implementation
    - *memoization*
    - *order of computation*
    - *dynamic programming*
16. time and space

Memoization

- store subproblem solutions in an array

For 0-1 knapsack, the subproblems are `knapsack(S',W')`.

- find a representation where *S'* and *W'* are integers ≥ 0
- if *W* and the weights w_i are integers, *W'* will be integer
- since the items can be considered in any order, let `S` be an array of all *n* items and *S'* = `S[k..n-1]`
  - *S'* can be represented by *k*

Formulation

Let `V[k][w]` = max value obtainable using items *k*..*n*-1 and a total weight ≤ *w*.

- initial subproblem
  - `V[0][W]`
- main case
  - $V[k][w] = \max\{ V[k+1][w-w_k]+v_k, V[k+1][w] \}$ if $w_k \le w$
  - $V[k][w] = V[k+1][w]$ otherwise
- base case
  - $V[k][0] = 0$ [no room left to take items]
  - $V[n][w] = 0$ [no items left to take, even with space]

Order of Computation

- $V[k][w] = \max\{ V[k+1][w-w_k]+v_k, V[k+1][w] \}$ if $w_k \le w$
- $V[k][w] = V[k+1][w]$ otherwise

Order of iteration –

- `V[k]` depends on `V[k+1]`
- fill in base cases first
  - $V[n][w] = 0$ [no items left to take, even with space]
- then fill in *k* from *n*-1 to 0
  - order doesn't matter for *w*

Dynamic Programming

```
for w = 0..W do
  V[n][w] = 0

for k = n-1..0 do
  for w = 0..W do
    if ( w_k <= w )
      V[k][w] = max(V[k+1][w-w_k]+v_k,V[k+1][w])
    else
      V[k][w] = V[k+1][w])
```

Time and Space

$V[k][w] = \max\{ V[k+1][w-w_k]+v_k, V[k+1][w] \}$ if $w_k \le w$

$V[k][w] = V[k+1][w]$ otherwise

Time and space –

- *Wn* entries to fill x O(1) per entry = O(*Wn*) total
  - may be much better than $O(2^n)$, depending on *W* (pseudopolynomial)
- *Wn* space required
- if the weights aren't integer, can solve to an arbitrary precision by multiplying *W* and w_i by a power of 10 – with a corresponding increase in time and space