Algorithm Design and Analysis

Dynamic Programming

Reminders

- **Midterm One is tonight at 7:00pm!!!**

Roadmap for today

- Learn about (hopefully review) ***dynamic programming***
- Try to solidify our understanding of how dynamic programming works
- Understand the two key elements:
  - Memoization
  - Optimal Substructure

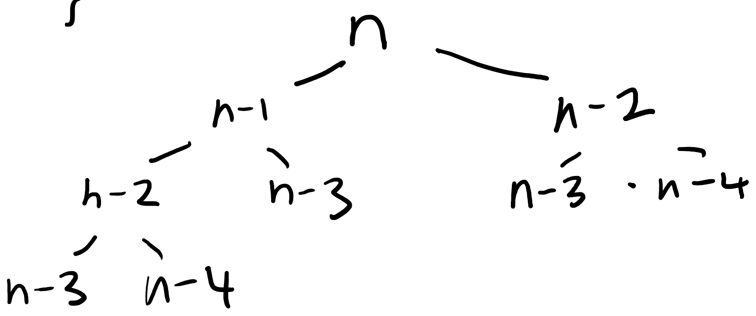
Key element #1: Memoization

Memoization: Don't solve the same problem twice

function fib(**int** n) {

  if n <= 1 **then** **return** 1
  else return fib(n-1) + fib(n-2)

}



dictionary<int,int> ***memo***

function fib(**int** n) {
  if n <= 1 **then** **return** 1
  if n **is in not in** ***memo*** **then** {

    memo[n] = fib(n-1) + fib(n-2)

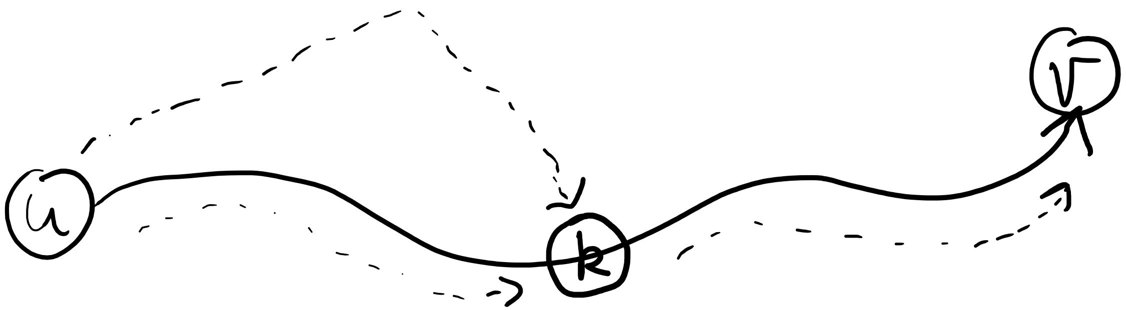
  }
  return   memo[n]

}

Key element #2: Optimal substructure

Optimal substructure: Break the problem into smaller versions of itself (recursively), and build the solution to the bigger problem by combining the answers to the smaller (sub-)problems

Example: shortest paths



“Recipe” for dynamic programming

1. ***Identify a set of optimal subproblems***
   - Write down a clear and unambiguous definition of the subproblems.
2. ***Identify the relationship between the subproblems***
   - Write down a recurrence that gives the solution to a problem in terms of its subproblems
3. ***Analyze the required runtime***
   - *Usually* (but not always) the number of subproblems multiplied by the time taken to solve a subproblem.

Often all that is required for a theoretical solution

---

4. ***Select a data structure to store subproblems***
   - *Usually* just an array. Occasionally something more complex.
5. ***Choose between bottom-up or top-down implementation***
6. ***Write the code!***

Only required if the answer is not “array”

Mostly ignored in this class (unless it’s a programming HW!)

Problems!

Longest Common Subsequence

Definition (Longest Common Subsequence): Given two strings, S of length n and T of length m. Produce the length of their longest common subsequence (not necessarily contiguous).

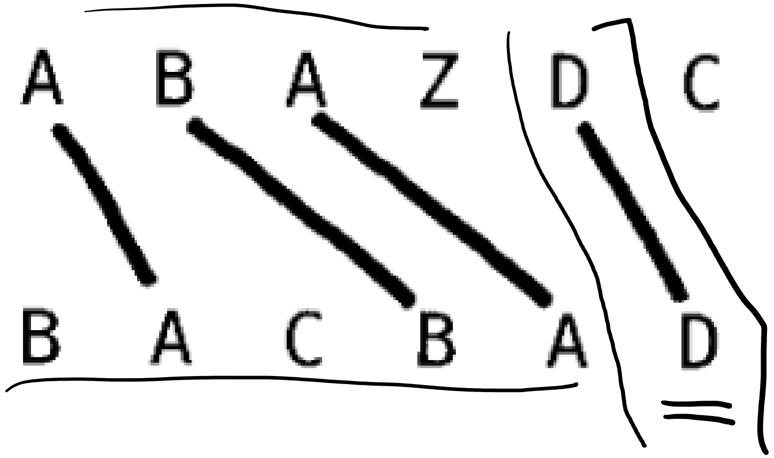
A B A Z D C

B A C B A D

A B A D

Longest Common Subsequence

Identify the optimal substructure / subproblems



Observation: *Optimal solution is a matching character + the longest common subsequence of the prefixes before them*

$LCS(i,j)$ = LCS of $S[\dots i]$ and $T[\dots j]$

Longest Common Subsequence

Relating the subproblems / deriving a recurrence

$$LCS(i,j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ \max(LCS(i-1,j), LCS(i,j-1)) & \\ 1 + LCS(i-1,j-1) & \text{if } S[i] = T[j] \end{cases}$$

Key idea!! We didn't know which option was best, so we tried both!

Longest Common Subsequence

Analysis: LCS can be solved in $O(nm)$ time using DP

subproblems = $O(nm)$

time per = $O(1)$

The Knapsack Problem

Definition (Knapsack): Given a set of $\boldsymbol{n}$ **items**, the i^{th} of which has **size** $\boldsymbol{s_i}$ and **value** $\boldsymbol{v_i}$. The goal is to find a subset of the items whose **total size is at most** $\boldsymbol{S}$, with **maximum possible value**.

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
size	3	4	2	6	7	3	5

$\boldsymbol{S = 15}$

The Knapsack Problem

Identify the optimal substructure / subproblems

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
size	3	4	2	6	7	3	5

Observation: *Optimal solution picks an item and then fills the remaining space optimally for that amount of space.*

Attempt #1: V(items S) = opt for items subset with cap S

Attempt #2: V(k, B) = opt for first k items with cap B

The Knapsack Problem

Relating the subproblems / deriving a recurrence

$$V(k,B) = \begin{cases} 0 & \text{if } k=0 \\ V(k-1,\ B) & \text{if } s_k > B \\ \max\left(\uparrow,\ V(k-1, B-s_k)+v_k\right) & \text{if } s_k \le B \end{cases}$$

Key idea again!! Try both options and take the best. "Clever brute force"

The Knapsack Problem

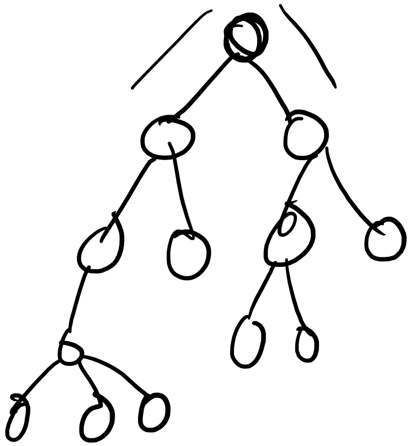
Analysis: Knapsack can be solved in $O(nS)$ time

subproblems = $O(nS)$

time per problem = 1

Independent sets on trees (Tree DP)

Definition (Independent set): Given a tree on $\boldsymbol{n}$ **vertices**, an independent set is a subset of the vertices $S \subseteq V$ such that none of them are adjacent. Each vertex has **a non-negative weight** $\boldsymbol{w_v}$, and we want to find the **maximum possible weight** independent set.



Observation: *Optimal solution either picks the root or doesn't, then it wants a max-weight independent set of its descendants.*

$w(v)$ = mwis of subtree rooted at v

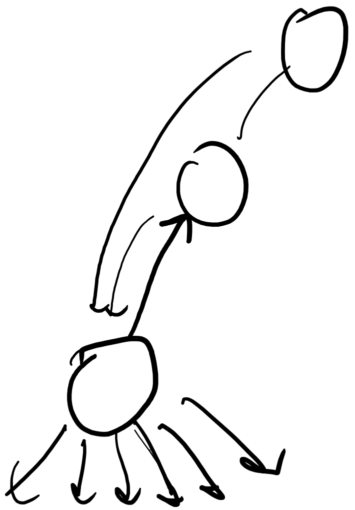
Independent sets on trees (Tree DP)

Relating the subproblems / deriving a recurrence

$$W(v) = \max \begin{cases} \sum_{u \in C(v)} W(u) & \text{(didn't pick } v\text{)} \\ \sum_{u \in GC(v)} W(u) + w_v & \text{(pick } v\text{)} \end{cases}$$

Independent sets on trees (Tree DP)

Analysis: MWIS on a tree can be solved in $O(n)$!!



$$O(n) + O(n) = O(n)$$

Take-home messages

- Breaking a problem into subproblems is hard. ***Common patterns:***
  - Can I use the first k elements of the input?
  - Can I restrict an integer parameter (e.g., knapsack size) to a smaller value?
  - On trees, can I solve the problem for each subtree? (Tree DP)
  - Many more on Thursday!
- Try a "***clever brute force***" approach.
  - Make one decision at a time and recurse, then take the best thing that results.
  - Can think of this as memorized backtracking
- Complexity analysis is *often* just subproblems $\times$ time per subproblem
  - But sometimes its harder and we must do some more analysis