

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 {  
  
    }  
    return  
}
```

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.

Only required if the answer is not “array”

5. *Choose between bottom-up or top-down implementation*

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

6. *Write the code!*

Problems!

Longest Common Subsequence

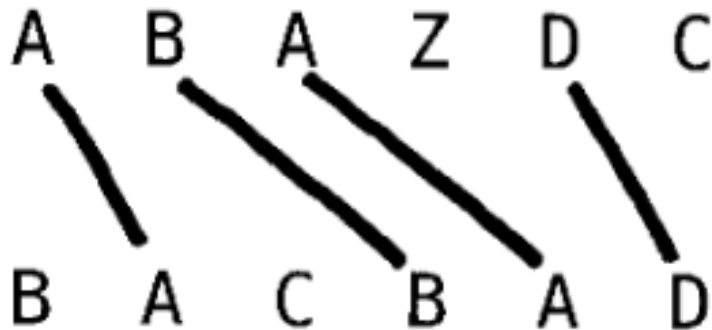
Definition (Longest Common Subsequence): Given two strings, S of length n and T of length m . Produce the length of their longest common subsequence (not necessarily contiguous).

A B A Z D C

B A C B A D

Longest Common Subsequence

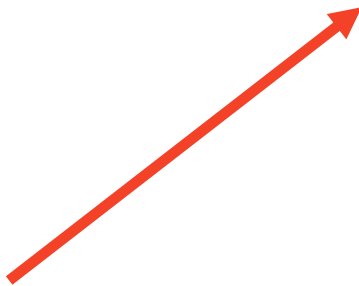
Identify the optimal substructure / subproblems



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

Longest Common Subsequence

Relating the subproblems / deriving a recurrence

$$LCS(i, j) = \left\{ \begin{array}{l} \text{...} \end{array} \right.$$


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

The Knapsack Problem

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

$$S = 15$$

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:

Attempt #2:

The Knapsack Problem

Relating the subproblems / deriving a recurrence

$$V(k, B) = \left\{ \begin{array}{l} \text{...} \end{array} \right.$$


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

Independent sets on trees (Tree DP)

Definition (Independent set): Given a tree on 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** 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.*

Independent sets on trees (Tree DP)

Relating the subproblems / deriving a recurrence

$$W(v) = \max \left\{ \begin{array}{l} \text{...} \\ \text{...} \end{array} \right.$$

Independent sets on trees (Tree DP)

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

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