Algorithm
Design and Analysis

Dynamic Programming

Roadmap for today

- Learn about (maybe review) ***dynamic programming***
- Understand the key elements:
  - Memoization
  - Optimal Substructure
  - Overlapping subproblems
- Practice a lot of DP problems!

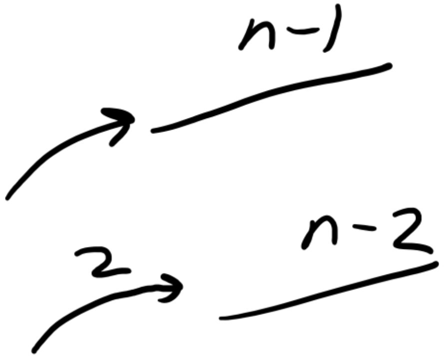
Starter example: Counting steps

You can climb up the stairs in increments of 1 or 2 steps.
How many ways are there to jump up n stairs?

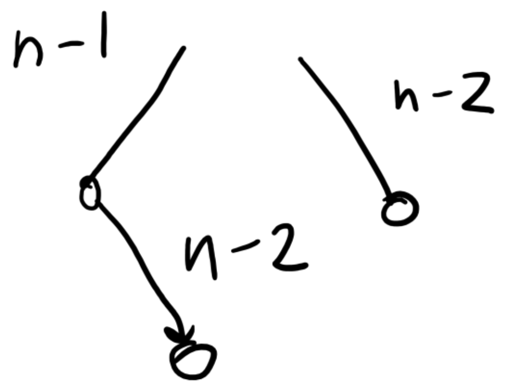
Could we solve this problem in terms of **smaller subproblems**?

#ways to climb $n-1$

$+$ #ways to climb $n-2$



Implementation #1




```
function stairs(int n) {
  if (n <= 1) then return 1
  else {
    let waysToTake1Step = stairs(n-1)
    let waysToTake2Steps = stairs(n-2)
    return waysToTake1Step + waysToTake2Steps
  }
}
```

Issue? Exponentially many recursive calls!!

Implementation #2

```
dictionary<int, int> memo

function stairs(int n) {
  if (n <= 1) then return 1
  if (n not in memo) {
    memo[n] = stairs(n-1)
              + stairs(n-2)
  }
  return memo[n]
}
```

Key Idea: Memoization
Don't solve the same problem twice! Store the result and reuse it!

Note: Memo dictionary
The memo dictionary does not need to be a hashtable! What should it be in this case?

When can we use DP?

- We could solve the stairs problem by using solutions to ***smaller*** instances of the stairs problem

```
stairs(n) = stairs(n-1) + stairs(n-2)
```

Key Idea: Optimal substructure

We say that a problem has *optimal substructure* if the optimal solution to the problem can be derived from optimal solutions to smaller instances (called ***subproblems***) of the problem.

When can we use DP?

- The DP implementation of stairs was faster because each subproblem was solved *only once* instead of *exponentially many times*

```
stairs(n) = stairs(n-1) + stairs(n-2)
```

Key Idea: **Overlapping subproblems**

Overlapping subproblems are subproblems that occur multiple (often exponentially many) times throughout the recursion tree. This is what distinguishes DP from ordinary recursion.

“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.
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!***

Often all that is required for a theoretical solution

Only required if the answer is not “array”

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

The Knapsack Problem

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	15	6	12
Size	3	4	2	6	7	3	5

$\boldsymbol{S = 15}$

Identifying Optimal Substructure

	A	B	C	D	E	F	G
Value	7	9	5	12	15	6	12
Size	3	4	2	6	7	3	5

Issue:

- How do we know whether to include a particular object X?
- We don't know in advance, so ***try both choices*** and pick best one!

Optimal substructure:

- Every object is either included or not included
- If an item X is included, the remaining S – Size(X) space is filled with some subset of the remaining items
- This is just a smaller instance of the knapsack problem!!

$V(k, B) :=$ value of best subset of $\{1 \ldots k\}$ with size at most B

Writing a recurrence

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

Key Idea: **Clever brute force**

We could not know in advance whether to include the i^{th} item or not, so we tried both possibilities and took the best one.

Analyzing the Runtime

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

$(n+1) \times (S+1)$ subproblems

$O(1)$ per subproblem

$\rightarrow$ $O(nS)$ time.

Max-weight independent set in a tree (Tree DP)

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.

Optimal substructure:

- A solution either includes the root or does not include the root
- If the root is chosen, the remaining solution is an independent set of the remaining vertices, excluding the root's children
- Each child/grandchild subtree is just another smaller instance of the MWIS-in-a-tree problem!!

$W(v) :=$ value of MWIS of the subtree rooted at v

Writing a Recurrence

$$W(v) = \max \begin{cases} \sum_{u \in Child(v)} W(u) & \text{(don't use } v\text{)} \\ \sum_{u \in GC(v)} W(u) + w_v & \text{(use } v\text{)} \end{cases}$$

Again: **Clever brute force**

We could not know in advance whether to include the root or not, so we tried both possibilities and took the best one!

Analyzing the Runtime

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

n subproblems

worst-case $O(n)$ time to solve subproblem

too pessimistic!!

O(degree) to solve a subproblem

$\rightarrow O(n)$ time in total

Longest Increasing Subsequence

Longest Increasing Subsequence

Definition (LIS): Given a sequence of n numbers $a_1, a_2, \ldots, a_n$, find the length of a longest strictly increasing subsequence.

- Note: A subsequence does not have to be contiguous

7	**0**	**4**	**3**	**10**	**11**	**17**	**15**

Defining Subproblems

7	0	4	3	10	11	17	15

Optimal substructure:

- An LIS ending with the element 15 extends the LIS that…

  - ended before position 15 ←
  - its value must be less than 15 ←

$\boxed{LIS(i) := \text{length of LIS ending with } a_i \text{ (must include } a_i\text{)}}$

Writing a Recurrence

$$LIS(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max\limits_{\substack{j<i \\ a_j < a_i}} LIS(j) + 1 & \end{cases}$$

Answer: $\max\limits_{i} LIS(i)$

Analyzing Runtime

$$\text{LIS}(i) = 1 + \max_{\substack{j \in [0,i) \\ a_j < a_i}} \text{LIS}(j)$$

- **Naïve runtime**: $O(n^2)$
- Can we do better?

- This recurrence is taking the ***maximum value in a range***
- Do we know a way to do this more efficiently?? SegTree

Optimized LIS: SegTree DP!

$$\mathrm{LIS}(i) = 1 + \max_{\substack{j \in [0,i) \\ a_j < a_i}} \mathrm{LIS}(j)$$

A:	**7**	**0**	**4**	**3**	**10**	**11**	**17**	**15**
SegTree:								

To be continued on Tuesday...

Optimized LIS: Pseudocode

```
function LIS(list A):
  n = length(A)
  results := SegTree(array of n+1 0's)
  sortedByVal := sorted list of (val, index) pairs
  for (val, index) in sortedByVal:



  return
```

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)
  - Can I solve the problem for a subset of the input (***next lecture, TSP***)
  - Can I keep track of more information (***next lecture, TSP***)
- 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 memoized backtracking
- Can I use a clever data structure to speed up the recurrence (SegTree DP!)
- Complexity analysis is *often* just subproblems $\times$ time per subproblem
  - But sometimes its harder and we must do some more analysis