Lecture 2: Dynamic Programming

- Learn/review Dynamic Programming
- Keys: Memoization, Optimal substructure, overlapping subproblems
- Example applications

Resources

- CLRS, *Introduction to Algorithms*, Chap 15/14 (3^{rd}/4^{th} ed.)
- Erikson, *Algorithms*, Chap 3
- CMU 15-451, Introduction to Algorithms, *Dynamic Programming I*

PRINCETON
COMPUTER SCIENCE

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?

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

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) {


  }
  return memo[n]
}
```

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

Note: Memo dictionary
Need only an array 90% times.

When can we use DP?

We solved 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 computed from optimal solutions to smaller instances (subproblems!) of the problem.

When can we use DP?

The DP implementation is faster because each subproblem is 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 (typically, exponentially!) 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 subproblems

2. **Identify the relationship between the subproblems**

   write down a recurrence that gives the solution to a problem in terms of subproblems

3. **Analyze the runtime**

   usually (but not always!) #subproblems × by time taken to solve a subproblem

4. **Select a data structure to store subproblems**

   usually an array. sometimes more sophisticated like a hashtable.

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

6. **Write the code!**

The *Knapsack* Problem

The *Knapsack* Problem

Input: a set of **n items**, the i^{th} of which has **size** s_i and **value** v_i.
Goal: find a subset of items with **total size** $\leq$ **S,** with **max value**.

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

$S = 15$

How would you solve the problem if you were allowed to pick fractional items?

The Fractional *Knapsack* Problem

Input: a set of **n items**, the i^{th} of which has **size** s_i and **value** v_i.
Goal: pick a fraction in [0,1] of each item with **total fractional size** $\leq$ **S,** with **max value**.

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

$S = 15$

Answer:

The *Knapsack* Problem

Input: a set of **n items**, the ith of which has **size** s_i and **value** v_i.
Goal: find a subset of items with **total size** $\leq$ **S,** with **max value**.

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

S = 15

The integral version is harder.
Unless P=NP, no polynomial time algorithm can exist (i.e., the problem is NP-hard).

The *Knapsack* Problem

Items	A	B	C	D	E	F	G
val	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 must try both choices and pick best one.

Optimal substructure:

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

Writing a recurrence

$$V(k, B) = \begin{cases} \\ \\ \\ \\ \end{cases}$$

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

Lemma: Our algorithm runs in time O(nS).

Why doesn't this contradict (or prove P=NP ☺) what we discussed earlier?

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

Independent sets on trees (Tree DP)

Definition (Independent Set)**:** An independent set in a graph G on vertex set V is a subset of vertices $S \subseteq V$ such that no pair has an edge between them.

Input: Tree on n vertices each with a non-negative weight w_v.
Goal: Find an independent set of vertices with max total weight.

Optimal substructure:

- A solution includes a root, or not.
- If the root is chosen, the remaining solution must exclude root's children (why?).
- Every child/grandchild subtree is just another smaller instance of MWIS-in-a-tree

Analyzing the runtime

Theorem: MWIS on a tree can be solved in O(n) time.

Longest Increasing Subsequence

Longest Increasing Subsequence

Input: A sequence of n numbers $a_1, a_2, \ldots, a_n$
Goal: find the longest strictly increasing subsequence.

Caution: 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…

Writing a recurrence

$$LIS(i) = \begin{cases} \\ \\ \\ \end{cases}$$

Answer:

Analyzing runtime

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

Naïve runtime:

Can we do better?

This recurrence is taking the ***maximum value in a range.*** ***Can we do this step more efficiently?***

Optimizing LIS

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

A:	7	0	4	3	10	11	17	15

Optimized LIS: Sortedness of best

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

$$= 1 + \max_{1 \le j \le t : best[j] < A[i]} j$$

$$S_j = \{p \mid LIS(p) = j\}$$

$$best[j] = \min S_j$$

$$E[j] : A[E[j]] = best[j]$$

Lemma: For every $1 \le j \le t$, $best[j] \le best[j+1]$

Why does this help?

Optimized LIS: binary search!

$$LIS(i) = 1 + \max_{1 \le j \le t : best[j] < A[i]} j$$

$$S_j = \{p \mid LIS(p) = j\}$$

$$best[j] = \min S_j$$

$$E[j] : A[E[j]] = best[j]$$

Lemma: For every $1 \le j \le t$, $best[j] \le best[j+1]$

→ Can compute the max above by binary search!

Optimized LIS: Pseudocode

```
function LIS(list A):
    Maintain t←0, best, E ←[]
    for 1 ≤ i ≤ n, do:
        s* ← BinSearch (best[1:t],A[i])
        j* ← E[s*] if s* ≠ 0, otherwise j* ← 0
        LIS[i] ← 1+s*
    if LIS[i] > t, then:
        t ← t+1
        best[t] ←A[i]
        E[t] ← I
    else if  A[i] < best[LIS[i]], then:
        best[LIS[i]] ←A[i]
        E[LIS[i]] ← i
return m = arg max LIS[1:n]
```

Optimized LIS: binary search!

Theorem: Optimized LIS runs in time O(n log n).

Key Takeaways

- Breaking a problem into subproblems is hard. **Common patterns:**
  - Use the first k elements of the input?
  - Restrict an integer parameter (e.g., knapsack size) to a smaller value?
  - (for trees), can I solve the problem on each subtree (Tree DP)
  - Solve the problem for a subset of inputs?
  - Keep track of more information?
- Try a *clever brute force* approach
  - Make one decision at a time and recurse, take the best thing that results.
  - Think of this as "memoized backtracking"
- Use a clever data structure to speed up recurrence (SegTree DP!)
- Complexity Analysis is typically: #subproblems x time per subproblem
  - But sometimes harder…