Lecture 16: Dyn. Prog. Subproblems

Dynamic Programming Review

- Recursion where subproblem dependencies **overlap**, forming DAG
- "Recurse but re-use" (Top down: record and lookup subproblem solutions)
- "Careful brute force" (Bottom up: do each subproblem in order)

Dynamic Programming Steps (SRT BOT)

1. **Subproblem** definition subproblem $x \in X$
   - Describe the meaning of a subproblem **in words**, in terms of parameters
   - Often subsets of input: prefixes, suffixes, contiguous substrings of a sequence
   - Often multiply possible subsets across multiple inputs
   - Often record partial state: add subproblems by incrementing some auxiliary variables
2. **Relate** subproblem solutions recursively $x(i) = f(x(j), \ldots)$ for one or more $j < i$
   - Identify a question about a subproblem solution that, if you knew the answer to, reduces the subproblem to smaller subproblem(s)
   - Locally brute-force all possible answers to the question
3. **Topological order** to argue relation is acyclic and subproblems form a DAG
4. **Base** cases
   - State solutions for all (reachable) independent subproblems where relation breaks down
5. **Original problem**
   - Show how to compute solution to original problem from solutions to subproblem(s)
   - Possibly use parent pointers to recover actual solution, not just objective function
6. **Time** analysis
   - $\sum_{x \in X} \text{work}(x)$, or if $\text{work}(x) = O(W)$ for all $x \in X$, then $|X| \cdot O(W)$
   - $\text{work}(x)$ measures **nonrecursive** work in relation; treat recursions as taking $O(1)$ time

Longest Common Subsequence (LCS)

- Given two strings A and B, find a longest (not necessarily contiguous) subsequence of A that is also a subsequence of B.
- Example: $A =$ `hieroglyphology`, $B =$ `michaelangelo`
- Solution: `hello` or `heglo` or `iello` or `ieglo`, all length 5
- Maximization problem on length of subsequence

1. **Subproblems**
   - $\boxed{x(i,j) = \text{length of longest common subsequence of suffixes } A[i:] \text{ and } B[j:]}$
   - For $0 \le i \le |A|$ and $0 \le j \le |B|$
2. **Relate**
   - Either first characters match or they don't
   - If first characters match, some longest common subsequence will use them
   - (if no LCS uses first matched pair, using it will only improve solution)
   - (if an LCS uses first in $A[i]$ and not first in $B[j]$, matching $B[j]$ is also optimal)
   - If they do not match, they cannot both be in a longest common subsequence
   - **Guess** whether $A[i]$ or $B[j]$ is not in LCS
   - $x(i,j) = \begin{cases} x(i+1,j+1)+1 & \text{if } A[i] = B[j] \\ \max\{x(i+1,j), x(i,j+1)\} & \text{otherwise} \end{cases}$
   - (draw subset of all rectangular grid dependencies)
3. **Topological order**
   - Subproblems $x(i,j)$ depend only on strictly larger i or j or both
   - Simplest order to state: Decreasing $i + j$
   - Nice order for bottom-up code: Decreasing i, then decreasing j
4. **Base**
   - $x(i, |B|) = x(|A|, j) = 0$ (one string is empty)
5. **Original problem**
   - Length of longest common subsequence of A and B is $x(0,0)$
   - Store parent pointers to reconstruct subsequence
   - If the parent pointer increases both indices, add that character to LCS

6. **Time**
    - # subproblems: $(|A| + 1) \cdot (|B| + 1)$
    - work per subproblem: $O(1)$
    - $O(|A| \cdot |B|)$ running time

```
def lcs(A, B):
    a, b = len(A), len(B)
    x = [[0] * (b + 1) for _ in range(a + 1)]
    for i in reversed(range(a)):
        for j in reversed(range(b)):
            if A[i] == B[j]:
                x[i][j] = x[i + 1][j + 1] + 1
            else:
                x[i][j] = max(x[i + 1][j], x[i][j + 1])
    return x[0][0]
```

Longest Increasing Subsequence (LIS)

- Given a string A, find a longest (not necessarily contiguous) subsequence of A that strictly increases (lexicographically).
- Example: $A =$ `carbohydrate`
- Solution: `abort`, of length 5
- Maximization problem on length of subsequence
- Attempted solution:
  - Natural subproblems are prefixes or suffixes of A, say suffix $A[i:]$
  - Natural question about LIS of $A[i:]$: is $A[i]$ in the LIS? (2 possible answers)
  - But then how do we recurse on $A[i+1:]$ and guarantee increasing subsequence?
  - Fix: add **constraint** to subproblems to give enough structure to achieve increasing property

1. **Subproblems**
   - $\boxed{x(i) = \text{length of longest increasing subsequence of suffix } A[i:] \text{ that includes } A[i]}$
   - For $0 \leq i \leq |A|$
2. **Relate**
   - We're told that $A[i]$ is in LIS (first element)
   - Next question: what is the *second* element of LIS?
     - Could be any $A[j]$ where $j > i$ and $A[j] > A[i]$ (so increasing)
     - Or $A[i]$ might be the *last* element of LIS
   - $x(i) = \max\{1 + x(j) \mid i < j < |A|, A[j] > A[i]\} \cup \{1\}$
3. **Topological order**
   - Decreasing i
4. **Base**
   - No base case necessary, because we consider the possibility that $A[i]$ is last
5. **Original problem**
   - What is the first element of LIS? **Guess!**
   - Length of LIS of A is $\max\{x(i) \mid 0 \leq i < |A|\}$
   - Store parent pointers to reconstruct subsequence

6. **Time**
    - # subproblems: $|A|$
    - work per subproblem: $O(|A|)$
    - $O(|A|^2)$ running time
    - Exercise: speed up to $O(|A| \log |A|)$ by doing only $O(\log |A|)$ work per subproblem, via AVL tree augmentation

```
def lis(A):
    a = len(A)
    x = [1] * a
    for i in reversed(range(a)):
        for j in range(i, a):
            if A[j] > A[i]:
                x[i] = max(x[i], 1 + x[j])
    return max(x)
```

Alternating Coin Game

- Given sequence of n coins of value $v_0, v_1, \ldots, v_{n-1}$
- Two players ("me" and "you") take turns
- In a turn, take first or last coin among remaining coins
- My goal is to maximize total value of my taken coins, where I go first
- First solution exploits that this is a **zero-sum game**: I take all coins you don't

1. **Subproblems**
   - Choose subproblems that correspond to the state of the game
   - For every contiguous subsequence of coins from i to j, $0 \leq i \leq j < n$
   - $\boxed{x(i,j) = \text{maximum total value I can take starting from coins of values } v_i, \ldots, v_j}$
2. **Relate**
   - I must choose either coin i or coin j **(Guess!)**
   - Then it's your turn, so you'll get value $x(i+1, j)$ or $x(i, j-1)$, respectively
   - To figure out how much value I get, subtract this from total coin values
   - $x(i,j) = \max\{v_i + \sum_{k=i+1}^{j} v_k - x(i+1,j), v_j + \sum_{k=i}^{j-1} v_k - x(i,j-1)\}$
3. **Topological order**
   - Increasing $j - i$
4. **Base**
   - $x(i,i) = v_i$
5. **Original problem**
   - $x(0, n-1)$
   - Store parent pointers to reconstruct strategy
6. **Time**
   - # subproblems: $\Theta(n^2)$
   - work per subproblem: $\Theta(n)$ to compute sums
   - $\Theta(n^3)$ running time
   - Exercise: speed up to $\Theta(n^2)$ time by precomputing all sums $\sum_{k=i}^{j} v_k$ in $\Theta(n^2)$ time, via dynamic programming (!)

- Second solution uses **subproblem expansion**: add subproblems for when you move next

1. **Subproblems**
   - Choose subproblems that correspond to the full state of the game
   - Contiguous subsequence of coins from i to j, and which player p goes next
   - $x(i, j, p)$ = maximum total value I can take when player $p \in \{\text{me}, \text{you}\}$ starts from coins of values $v_i, \ldots, v_j$
2. **Relate**
   - Player p must choose either coin i or coin j **(Guess!)**
   - If $p =$ me, then I get the value; otherwise, I get nothing
   - Then it's the other player's turn
   - $x(i, j, \text{me}) = \max\{v_i + x(i+1, j, \text{you}), v_j + x(i, j-1, \text{you})\}$
   - $x(i, j, \text{you}) = \min\{x(i+1, j, \text{me}), x(i, j-1, \text{me})\}$
3. **Topological order**
   - Increasing $j - i$
4. **Base**
   - $x(i, i, \text{me}) = v_i$
   - $x(i, i, \text{you}) = 0$
5. **Original problem**
   - $x(0, n-1, \text{me})$
   - Store parent pointers to reconstruct strategy
6. **Time**
   - # subproblems: $\Theta(n^2)$
   - work per subproblem: $\Theta(1)$
   - $\Theta(n^2)$ running time

Subproblem Constraints and Expansion

- We've now seen two examples of constraining or expanding subproblems
- If you find yourself lacking information to check the desired conditions of the problem, or lack the natural subproblem to recurse on, try subproblem constraint/expansion!
- More subproblems and constraints give the relation more to work with, so can make DP more feasible
- Usually a trade-off between number of subproblems and branching/complexity of relation
- More examples next lecture

6.006	Lecture 16	Apr. 16, 2020

TODAY: Dynamic Programming II (of 4)
- multiple seqs. - substring subprobs. - parent pointers
- subproblem constraints & expansion
- examples → Longest Common Subsequence (LCS)
  → Longest Increasing Subsequence (LIS)
  → Alternating Coin Game

Recall: SRTBOT paradigm for recursive alg. design & (with memoization) for DP alg. design
- Subproblem definition
  - for sequence S, try → prefixes S[:i] $\Theta(n)$
    → suffixes S[i:] $\Theta(n)$
    → substrings S[i:j] $\Theta(n^2)$
- Relate subproblem solutions recursively
  - identify question about subproblem solution that, if you knew answer, reduces to "smaller" subprob(s)
  - locally brute-force all answers to that question
  - can think of correctly guessing answer, then loop
- Topological order on subproblems to guarantee acyclic relation
  → subproblem/call DAG (a) → (b) ≡ b needs a
- Base cases of relation
- Original problem: solve via subproblem(s)
- Time analysis $\leq \sum_{\text{subprob.}}$ nonrec. work in relation
  ≤ # subproblems · nonrecursive work in relation

Longest Common Subsequence (LCS):

given two sequences A & B, find longest sequence L that's a subsequence of both A & B

sequential but not necessarily contiguous

- e.g.: HIEROGLYPHOLOGY
  vs. MICHAELANGELO
  } → HELLO

Subproblems for multiple inputs: trick
- multiply subproblem spaces (e.g. suffixes × suffixes)
- still polynomial for O(1) inputs (e.g. sequences)

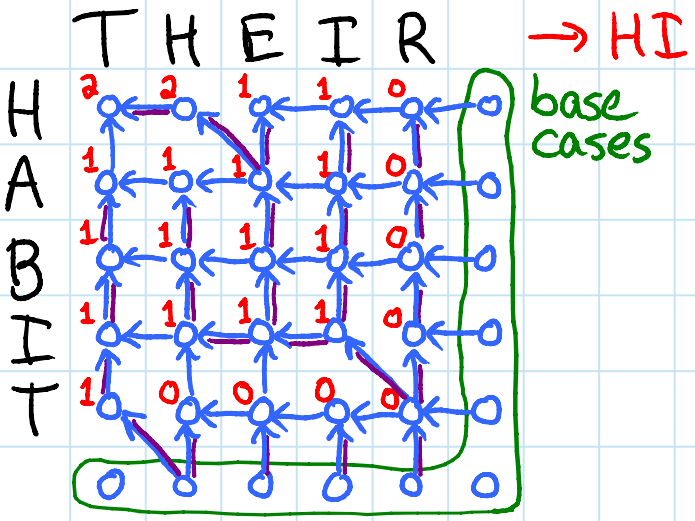
SRTBOT:
- Subproblems: $L(i,j) = LCS(A[i:], B[j:])$
  for $0 \le i \le |A|$ & $0 \le j \le |B|$
  $\Rightarrow (|s|+1)\cdot(|t|+1) = \Theta(|s|\cdot|t|)$ subproblems
- Relate: $L(i,j) =$ if $A[i] == B[j]$: $1 + L(i+1, j+1)$
  else: $\max\{L(i+1,j), L(i,j+1)\}$
  - if first symbols don't match, one of them isn't in LCS ~ guess which one
  - if match, claim a LCS pairs them up:
    - if LCS uses exactly one of them, say it pairs up $A[i]$ with $B[>j]$, then doesn't use $B[j]$ so could instead pair $A[i]$ with $B[j]$
    - if LCS uses neither, contradiction (could add a pair)

- Topo. order: for $i = |A|, \ldots, 0$: for $j = |B|, \ldots, 0$:
  OR decreasing $i + j$
- Base cases: $L(|A|, j) = L(i, |B|) = \emptyset$
- Original: $L(0, 0)$
- Time: $\Theta(|A| \cdot |B|)$ ~ $\Theta(1)$ time / subproblem

Subproblem DAG:

$L(i, j)$

parent pointers

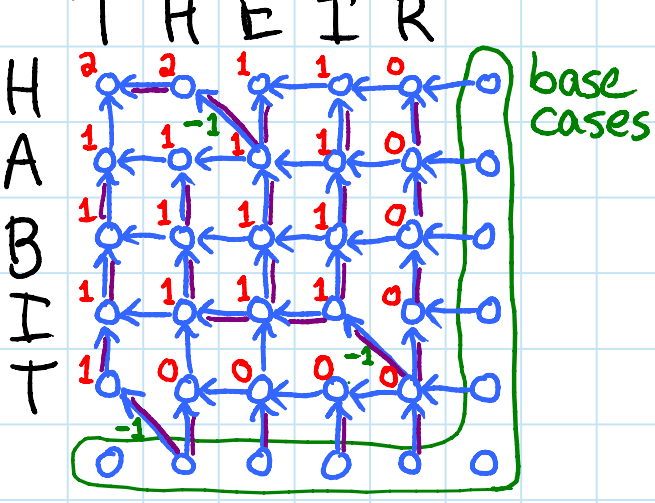


Recovering the sequence:

- remember the winning choice in relation
- here reduce to single smaller subprobem
  ⇒ get parent pointers

- equivalent to weighted shortest path from a base case to original in subproblem DAG:
  - diagonals weight -1
  - other edges $\emptyset$

(like bowling, but unlike Fibonacci)



Longest Increasing Subsequence (LIS):

given a sequence A, find longest subsequence that strictly increases

- e.g. C A R B O H Y D R A T E → A B O R T

Attempt:

- Subproblems: $L(i) = LIS(A[i:])$
- Relate: $L(i) \stackrel{?}{=} \max\{L(i+1), 1+L(i+1) \cdots ?\}$ ← don't include L[i] ← include L[i]
  - how can we guarantee increasing subseq.?
  - no way to constrain smaller subproblem...

Subproblem constraints: solve a more specific problem (add constraints) to enable relation

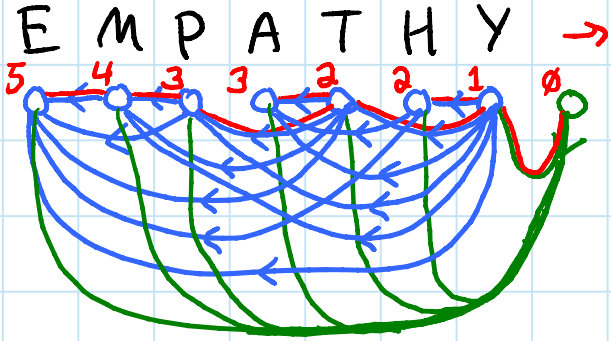
Solution:

- Subproblems: $L(i)$ = LIS of $A[i:]$ that constraint → starts with (includes) $A[i]$
- Relate: $L(i) = \max\{1+L(j) \mid i < j \le |A|, A[i] < A[j]\} \cup \{1\}$
  - $A[i] < A[j]$: guarantees increasing
  - $\{1\}$: increase impossible

- nonconstant branching (like DAG SPs but unlike other DPs so far)

- Topo. order: for $i = |A|, \ldots, \emptyset$
- Base case: $L(|A|) = \emptyset$
- Original: $\max\{L(i) \mid 0 \le i < |A|\}$
  - where to start? guess / brute-force options
- Time: $\Theta(n)$ subprobs. $\cdot$ $\Theta(n)$ nonrec. work in relation
  $= \Theta(n^2)$

Subproblem DAG:

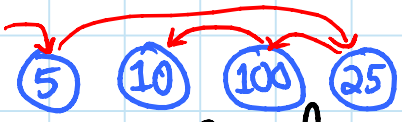


- equivalent to longest path in this DAG
  = shortest path with weights -1

- as before, can store parent pointers

Exercise: speed up LIS DP to $O(n \lg n)$
by computing max in relation
via memo table data structure

Alternating coin game:



- given sequence of n coins of value $v_0, v_1, \ldots, v_{n-1}$
- 2 players ("me" & "you") take turns
- in a turn, take first or last coin among remaining
- goal: maximize total value of my taken coins where I go first

What if I went second?

$\sum_{i=0}^{n-1} v_k$ − above

above: what you take; $\sum_{i=0}^{n-1} v_k$: what remains

Solution 1:

- Subproblems: $X(i, j)$ = max. value playing from coins of value $v_i, v_{i+1}, \ldots, v_j$ for $0 \le i \le j < n$
- Relate: $X(i,j) = \max\{ v_i + \sum_{k=i+1}^{j} v_k - X(i+1, j),$ ← take i

  $v_j + \sum_{k=i}^{j-1} v_k - X(i, j-1) \}$ ← take j

  (zero-sum game)

  $\sum_{k=i+1}^{j} v_k$: what you take; $X(i+1, j)$: what remains

  $\sum_{k=i}^{j-1} v_k$: what you take; $X(i, j-1)$: what remains
- Topo. order: increasing $j - i$
- Base cases: $X(i, i) = v_i$
- Original: $X(0, n-1)$
- Time: $\Theta(n^2)$ subproblems · $\Theta(n)$ nonrec. work $= \Theta(n^3)$

Exercise: speed up to $\Theta(n^2)$ time by precomputing all $\sum_{k=i}^{j} v_k$ in $\Theta(n^2)$ time via DP!

Solution 2: separate subproblems for when you are next to move

- one of my moves naturally leads to this
- Subproblems: $X(i, j, p)$ = my max. value playing from coins of value $v_i, \ldots, v_j$ where player p moves first, for $0 \le i \le j < n$ & $p \in \{me, you\}$
- Relate: $X(i, j, me) = \max\{v_i + X(i+1, j, you), \leftarrow$ I take i
  $v_j + X(i, j-1, you)\} \leftarrow$ I take j
  $X(i, j, you) = \min\{X(i+1, j, me), \leftarrow$ you take i
  $X(i, j-1, me)\} \leftarrow$ you take j
  adversarial opponent →
- Base: $X(i, i, me) = v_i$; $X(i, i, you) = \emptyset$
- Original: $X(0, n-1, me)$
- Time: $\Theta(n^2)$ subprobs. · $\Theta(1)$ nonrec. work $= \Theta(n^2)$ — cleaner DP & faster!

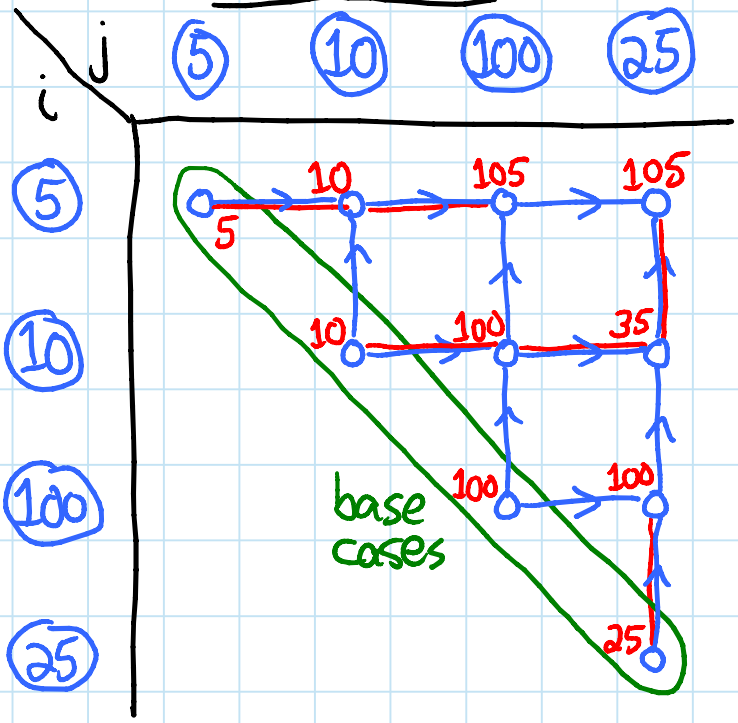
Subproblem expansion: adding subproblems (usually with extra constraints) gives relation more to work with, so can make DP more feasible

- if lacking enough information to check desired conditions, try subproblem constraints / expansion
- trade-off between # subproblems & branching/complexity of relation

→ More examples next lecture!

Subproblem DAGs:

Solution 1



Solution 2