Parallel And Sequential Data Structures and Algorithms

Dynamic Programming II

Learning Objectives

- Practice more examples of **dynamic programming**
- The **Knapsack** Problem
- The **Edit Distance** Problem
- (Yet Another) **Parenthesis** Problem

Review: Coin Change

Step 1 of DP: Define Subproblems

Step 1 of any dynamic programming algorithm is **always** to **define your set of subproblems**, precisely and unambiguously

Define, for all $0 \leq v \leq V$:

$$\text{Possible}(v) := \begin{cases} \textbf{True} & \text{if exactly } \$v \text{ can be made} \\ \textbf{False} & \text{otherwise} \end{cases}$$

- Give your subproblems reasonable names
- Clearly state the value of the subproblem (e.g., True/False)
- Describe the meaning of the parameters (e.g., v)
- Give the domain of the parameters (e.g., $0 \leq v \leq V$)

Step 2 of DP: Recursive Solution

Step 2 of any dynamic programming algorithm is to solve a subproblem by writing a recursive definition of the solution in terms of solutions to smaller subproblems. We write this as a **recurrence relation**.

- With coins $c = [2,5,10]$, we know that (roughly)

$$\text{Possible}(v) = \text{Possible}(v-2) \textbf{ or } \text{Possible}(v-5) \textbf{ or } \text{Possible}(v-10)$$

$$\text{Possible}(v) \coloneqq \begin{cases} \textbf{True} & \text{if } v = 0 \\ \bigvee_{\substack{i \in [n] \\ c_i \le v}} \text{Possible}(v - c_i) & \text{otherwise} \end{cases}$$

Bottom-up Dynamic Programming

Definition (Bottom-Up DP): Bottom-up DP solves the subproblems **in order from smallest to largest** (so that no subproblem is solved twice).

```
fun coin_change(c : sequence<int>, V : int) -> bool:
  possible = [False for v in 0...V]
  for v in 0...V:
    if v == 0:
      possible[v] ← True
    else:
      answer = False
      for denomination in c:
        if denomination <= v:
          answer ← answer or possible[v - denomination]
      possible[v] ← answer
  return possible[V]
```

Top-down Dynamic Programming

Definition (Top-Down DP): **Top-down** DP solves the subproblems recursively but keeps a cache of previously solved subproblems.

```
fun coin_change(c : sequence<int>, V : int) -> bool:
  memoized = [⊥ for v in 0...V]
  fun possible(v):
    if memoized[v] == ⊥:
      if v == 0: memoized[v] ← True
      else:
        answer = False
        for denomination in c:
          if denomination <= v:
            answer ← answer or possible(v - denomination)
        memoized[v] ← answer
      return memoized[v]
  return possible(V)
```

Now with base case included!

General DP Design Pattern

1. **Reduce the problem to smaller versions of itself**
   - If I make a choice in the solution, what smaller problem remains?
2. **Define subproblems precisely**
   - Identify a set of parameters that describe the smaller problems.
3. **Derive a recurrence relation**
   - Express the solution to each subproblem in terms of solutions to smaller subproblems
4. **Analyze the cost of the solution**
   - For most problems, the cost is the number of subproblems multiplied by the cost of solving a single subproblem

The Knapsack Problem

The Knapsack Problem

Problem (0-1 Knapsack): We are given n items. Each item i has a size s_i and a value v_i. We are also given a capacity S. The goal is to select a subset of the items with total size at most S that maximizes total value. Each item may be used at most once

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

$S = 15$

Items: $\{A, B, F, G\}$
Size: 15
Value: 34

Knapsack Subproblems

Subproblem Design:

- Consider making a **choice** about the solution
- Making a choice should make the **problem smaller**
- What does a smaller problem look like? Those are your **subproblems**
- Your recurrence then tries all options for the choice

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

$S = 15$

Choice: Do I take Item G?

YES → Solve new knapsack: Items: {A,B,C,D,E,F} Capacity: $S = 10$

NO → Solve new knapsack: Items: {A,B,C,D,E,F} Capacity: $S = 15$

Knapsack Subproblems

- When we choose to add an item, we can **no longer consider that item**, and the **capacity (S) decreases**

Define, for all $0 \le k \le n$, and $0 \le B \le S$:

$$\text{MaxVal}(k, B) \coloneqq \begin{cases} \text{maximum value achievable using} \\ \text{items } [0, \dots, k) \text{ with capacity } B \end{cases}$$

- i.e., our subproblems consider just a **prefix of the items** with **a lowered capacity**.

Knapsack Recurrence

Key Idea: Make the choice of whether to include item $k-1$. Try including it and not including it, then take the best of the two outcomes

$$\text{MaxVal}(k, B) = \begin{cases} 0 & \text{if } k = 0 \\ \text{MaxVal}(k-1, B) & \text{if } s_{k-1} > B \\ \max(\text{MaxVal}(k-1, B),\ v_{k-1} + \text{MaxVal}(k-1, B - s_{k-1})) & \text{otherwise} \end{cases}$$

Don't take item $k-1$

Value of taking item $k-1$

Taking item $k-1$ **reduces capacity**

DP = "Clever Brute Force"

- The key part of the recurrence was the "choice"
- We try taking item $k-1$, or not taking, then take the best outcome

$$\max\left(\mathrm{MaxVal}(k-1, B), v_{k-1} + \mathrm{MaxVal}(k-1, B - s_{k-1})\right)$$

Clever brute force: This algorithm is correct because it tries **every subset of items**. This recurrence implemented without memoization is literally an exponential-time brute force over all subsets of items. DP turns an exponential-time brute force algorithm into an efficient algorithm by systematically eliminating redundancies in that search.

Analysis of Knapsack

Theorem (Cost of Knapsack DP): Given n items and a capacity of S, the Knapsack DP costs $O(nS)$ work.

- *There are* $(n+1)(S+1) = O(nS)$ *subproblems*
- *Each does* $O(1)$ *work: try taking item* k *versus not taking it*
- *Therefore, the total cost is* $O(nS)$ *work*

Knapsack: Example Code

$$\text{MaxVal}(k, B) = \begin{cases} 0 & \text{if } k = 0 \\ \text{MaxVal}(k-1, B) & \text{if } s_{k-1} > B \\ \max\big(\text{MaxVal}(k-1, B), v_{k-1} + \text{MaxVal}(k-1, B - s_{k-1})\big) & \text{otherwise} \end{cases}$$

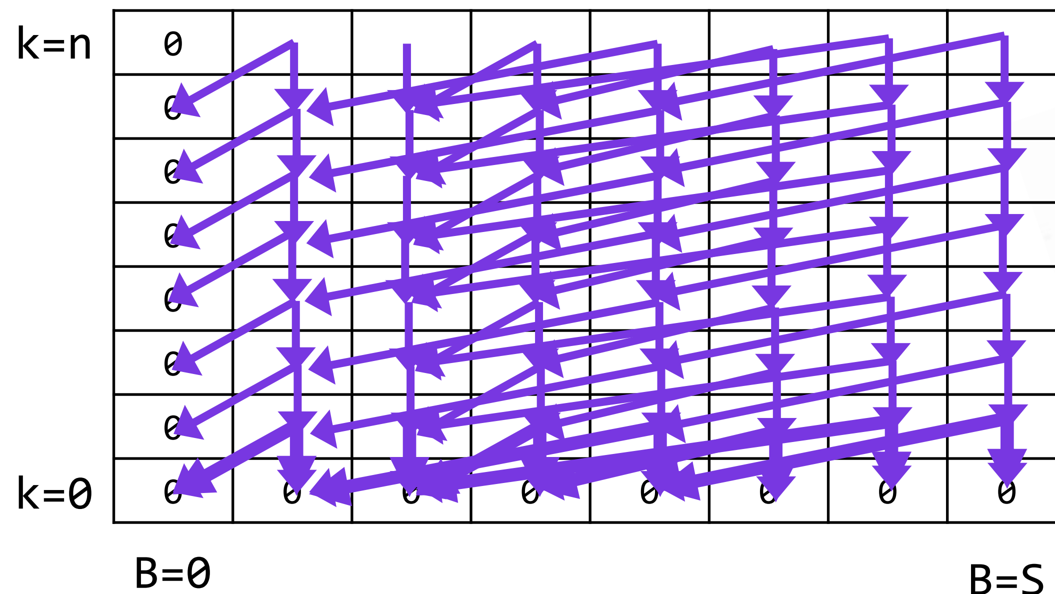
```
fun knapsack(s : sequence<int>, v : sequence<int>, S : int) -> int:
  n = |s|
  MaxVal = [[0 for _ in 0...S] for _ in 0...n]
  for k in 1...n:
    for B in 1...S:
      if s[k-1] > B:
        MaxVal[k][B] ← MaxVal[k-1][B]
      else:
        MaxVal[k][B] ← max(MaxVal[k-1][B], v[k-1] + MaxVal[k-1][B-s[k-1]])

  return MaxVal[n][S]
```

Base cases initialized (don't need to consider in loop)

Where's the Parallelism??

- Observe that $\mathrm{MaxVal}(k, B)$ depends only on $\mathrm{MaxVal}(k-1, B)$ and $\mathrm{MaxVal}(k-1, B-s_{k-1})$
- In other words, if we think of filling a 2D table of $\mathrm{MaxVal}(k, B)$, each row depends only on the previous row!

The contents of a row can be computed **in parallel**!



Parallel Knapsack

```
fun knapsack(s : sequence<int>, v : sequence<int>, S : int) -> int:
  n = |s|
  MaxVal = parallel [[0 for _ in 0...S] for _ in 0...n]
  for k in 1...n:
    MaxVal[k] ← tabulate((fn (B : int):
      if B == 0: return 0
      else if s[k-1] > B: return MaxVal[k-1][B]
      else: return max(MaxVal[k-1][B], v[k-1] + MaxVal[k-1][B-s[k-1]])
  ), S+1)
  return MaxVal[n][S]
```

Theorem (Cost of Parallel Knapsack): Given n items and a capacity of S, Parallel Knapsack costs $O(nS)$ work and $O(n)$ span.

The Edit Distance Problem

Minimum Edit Distance

Problem (Edit Distance): Given strings S and T of length n and m respectively, what is the minimum number of edit operations required to transform S into T? We allow the following edit operations:

- **Insertion**: insert a character into the string,
- **Deletion**: delete a character from the string,
- **Substitution**: replace one character with another

- E.g., how many operations to transform `"kitten"` into `"sitting"`?

KITTEN → **SITTEN** → **SITTIN** → **SITTING**

Edit Distance Subproblems

Subproblem Design:

- Consider making a **choice** about the solution
- Making a choice should make the **problem smaller**
- What does a smaller problem look like? Those are your **subproblems**
- Your recurrence then <u>tries all options</u> for the choice

- **Important observation**: Order doesn't matter

KITTEN → **<u>S</u>ITTEN** → **SITT<u>I</u>N** → **SITTIN<u>G</u>**

- I could have done these three operations in any order…
- Let's make the last character match (**<u>G</u>**) and **choose how**

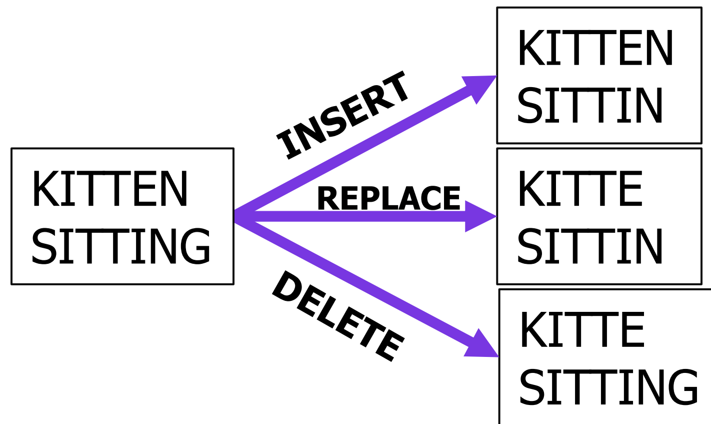
Edit Distance Subproblems

KITTEN → SITTEN → SITTIN → SITTING

- Let's choose to make the last character match (**G**)
- There are **three ways** to do so

Insert G	Replace N With G	Match G Earlier (Delete N)
KITTENG SITTING	KITTEN SITTING	KITTEN SITTING
Compute edit distance: KITTEN SITTIN	Compute edit distance: KITTE SITTIN	Compute edit distance: KITTE SITTING

Edit Distance Subproblems



- Each choice, we remove one character from one or both strings
- This suggests subproblems corresponding to **prefixes**

Define, for all $0 \le i \le n$, and $0 \le j \le m$:

$$\text{MED}(i, j) \coloneqq \begin{cases} \text{minimum edit distance between} \\ S[0 \dots i) \text{ and } T[0 \dots j) \end{cases}$$

Edit Distance Recurrence

Key Idea: Choose between inserting the last character of T, replacing the last character of S with the last character of T or deleting the last character of S. Take the best of the three choices

$$\text{MED}(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min\left(\begin{cases} \text{MED}(i-1,j-1) + (1 \textbf{ if } S[i-1] \neq T[j-1] \textbf{ else } 0) \\ \text{MED}(i-1,j) + 1 \\ \text{MED}(i,j-1) + 1 \end{cases}\right) & \text{otherwise} \end{cases}$$

Insert $T[j-1]$

Delete $S[i-1]$

Replace $S[i-1]$ with $T[j-1]$. This costs 0 if the characters are already the same

Analysis of Edit Distance

Theorem (Cost of Edit Distance DP): Given string S and T of lengths n and m, edit distance can be solved in a cost of $O(nm)$ work.

- *There are* $(n+1)(m+1) = O(nm)$ *subproblems*
- *Each does* $O(1)$ *work: try three operations and take the min*
- *Therefore, the total cost is* $O(nm)$ *work*

Parallelism: Edit distance *can* be parallelized, but its tricker than knapsack, because subproblems depend on other subproblems in the same row and same column. Think about it as an exercise (or read the notes).

Edit Distance: Example Code

$$\text{MED}(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min\left(\begin{cases} \text{MED}(i-1,j-1) + (1 \textbf{ if } S[i-1] \neq T[j-1] \textbf{ else } 0) \\ \text{MED}(i-1,j) + 1 \\ \text{MED}(i,j-1) + 1 \end{cases}\right) & \text{otherwise} \end{cases}$$

```
fun edit_distance(S : sequence<char>, T : sequence<char>) -> int:
  n = |S|, m = |T|
  MED = [[i+j for j in 0...m] for i in 0...n]
  for i in 1...n:
    for j in 1...m:
      MED[i][j] ← min(
        MED[i-1][j-1] + (1 if S[i-1] != T[j-1] else 0),
        MED[i-1][j] + 1,
        MED[i][j-1] + 1
      )
  return MED[n][m]
```

Base cases initialized (don't need to consider in loop)

(Yet Another) Parenthesis Problem

Boolean Parenthesisation

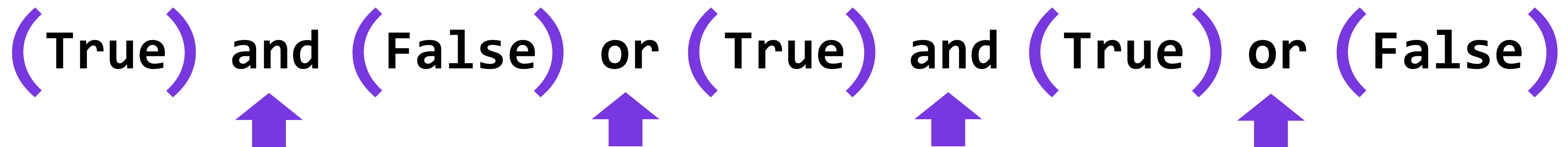
Problem (Boolean Expression Feasibility): You are given a Boolean expression consisting of the literals `True` and `False`, connected by the binary operators AND (∧) and OR (∨). The expression is written without parentheses. Is it possible to fully **parenthesize the expression** so that **it evaluates to True**?

`((True and (False or True)) and (True or False))`

Boolean Parenthesisation Subproblems

Subproblem Design:

- Consider making a **choice** about the solution
- Making a choice should make the **problem smaller**
- What does a smaller problem look like? Those are your **subproblems**
- Your recurrence then tries all options for the choice



- **Choice**: Choose the **final operator**
- Then recursively parenthesize both sides

Boolean Parenthesisation Subproblems

```
True  and  (False  or  True  and  True)  or  False
            i                       j
```

Define, for all $0 \leq i < n$, and $i \leq j < n$:

$$\text{CanParen}(i, j) := \begin{cases} \textbf{True} & \text{if the subexpression from } b_i \text{ to } b_j \text{ can be parenthesized to evaluate to } \textbf{True} \\ \textbf{False} & \text{otherwise} \end{cases}$$

Parameters: Knapsack and Edit distance both used prefixes of the input as subproblems. When this is not enough information, considering all **contiguous subsequences** of the input is often a good subproblem choice.

Boolean Parenthesisation Recurrence

Key Idea: Choose which operator comes last and recursively parenthesize the left and right subexpressions.

$$\text{CanParen}(i,j) = \begin{cases} b_i & \text{if } i = j \\ \displaystyle\bigvee_{i \le k < j} \begin{cases} \text{CanParen}(i,k) \land \text{CanParen}(k+1,j) & \text{if op}_k = \land \\ \text{CanParen}(i,k) \lor \text{CanParen}(k+1,j) & \text{if op}_k = \lor \end{cases} & \text{otherwise} \end{cases}$$

Try all final operators
(split points)

Boolean Parenthesisation Analysis

Theorem (Cost of Boolean Parenthesisation DP): Given a Boolean expression with n literals, the Boolean expression feasibility problem can be solved in $O(n^3)$ work

- *There are* $\Theta(n^2)$ *subproblems*
- *Subproblem* (i, j) *does* $(j - i)$ *work, which is* $O(n)$
- *So, the total work is* $O(n^3)$

Parallelism: This problem can also be parallelized. Think about which subproblems can be solved in parallel and identify any other opportunities for parallelism as an exercise (or read the notes).

Boolean Parenthesisation: Code

Order: Subproblems are ordered by **length**.

```
fun canParenthesize(b : sequence<bool>, op : sequence<char>) -> bool:
  n = |b|
  CanParen = [[False for _ in 0...n-1] for _ in 0...n-1]
  for i in 0...n-1: CanParen[i][i] ← b[i]
  for length in 0...n-1:
    for i in 0...(n-length)-1:
      j = i + length
      answer = False
      for k in i...j-1:
        if op[k] == '∧': answer ← answer or (CanParen[i][k] and CanParen[k+1][j])
        else: answer ← answer or (CanParen[i][k] or CanParen[k+1][j])
      CanParen[i][j] ← answer
  return CanParen[0][n-1]
```

Summary

- **Subproblem Design:**
  - Consider making a **choice** about the solution
  - Making a choice should make the **problem smaller**
  - What does a smaller problem look like? Those are your **subproblems**
  - Your recurrence then tries all options for the choice
- **Common Subproblem Parameter Patterns**:
  - Reduce to smaller value (e.g., Coin Change value, Knapsack capacity)
  - Reduce to prefix of the input (e.g., Knapsack items, Edit Distance)
  - Reduce to an interval of the input (e.g., Boolean Parenthesisation)