Agenda

- Contextualization of Dynamic Programming
- Main features
  - Subproblem overlapping
  - Principle of optimality
- Approaches
  - Memoization (Top-Down)
  - Tab (Bottom-up)
Context
Dynamic programming

- It is a powerful algorithm design technique
Dynamic programming

- It is a powerful algorithm design technique
- Two perspectives on PD:
  - DP ≈ "careful brute force"
  - Using intelligently, one can reduce "exponential" problems to polynomials
Context
Dynamic programming

- It is a powerful algorithm design technique
- Two perspectives on PD:
  - DP ≈ "careful brute force"
  - Using intelligently, one can reduce "exponential" problems to polynomials
  - DP ≈ Recursion + "reuse"
  - We will be more precise throughout the class
Bellman, (1984) p. 159 explained that he invented the name “dynamic programming” to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who “had a pathological fear and hatred of the term, research.” He settled on “dynamic programming” because it would be difficult give it a “pejorative meaning” and because “It was something not even a Congressman could object to.

[John Rust 2006]
[https://editorialexpress.com/jrust/research/papers/dp.pdf]
Bellman, (1984) p. 159 explained that he invented the name “dynamic programming” to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who “had a pathological fear and hatred of the term, research.” He settled on “dynamic programming” because it would be difficult give it a “pejorative meaning” and because “It was something not even a Congressman could object to.

[John Rust 2006]  
[https://editorialexpress.com/jrust/research/papers/dp.pdf]
Dynamic programming

Features
  ○ Overlapping problems (??)
  ○ Principle of optimality (??)
Fibonacci sequence

- Recurrence:
  - $F_n = F_{n-1} + F_{n-2}$

- Base case:
  - $F_1 = F_2 = 1$, or
  - $F_0 = F_1 = 1$
Fibonacci sequence

▶ Recurrence:
  ○ $F_n = F_{n-1} + F_{n-2}$

▶ Base case:
  ○ $F_1 = F_2 = 1$, or
  ○ $F_0 = F_1 = 1$

<table>
<thead>
<tr>
<th></th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</th>
<th>$F_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Fibonacci sequence

- Recurrence:
  - $F_n = F_{n-1} + F_{n-2}$

- Base case:
  - $F_1 = F_2 = 1$, or
  - $F_0 = F_1 = 1$

<table>
<thead>
<tr>
<th></th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</th>
<th>$F_4$</th>
<th>$F_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
Fibonacci sequence

- **Recurrence:**
  - $F_n = F_{n-1} + F_{n-2}$

- **Base case:**
  - $F_1 = F_2 = 1$, or
  - $F_0 = F_1 = 1$

- **Goal:**
  - Compute $F_n$
Fibonacci sequence

Naive solution

1. `def fib(n):`
2.   `if n <= 2:
3.       f = 1`
4.   `else:
5.       f = fib(n-1) + fib(n-2)
6.   `return f`
Fibonacci sequence
Naive solution

```python
1. def fib(n):
2.     if n <= 2:
3.         f = 1
4.     else:
5.         f = fib(n-1) + fib(n-2)
6.     return f
```
1. def fib(n):
2.   if n <= 2:
3.     f = 1
4. else:
5.     f = fib(n-1) + fib(n-2)
6. return f
Fibonacci sequence
Naive solution

1. ```def fib(n):
2.     if n <= 2:
3.         f = 1
4.     else:
5.         f = fib(n-1) + fib(n-2)
6.     return f```

▷ Does the algorithm work?
▷ Is it a good algorithm?
Fibonacci sequence
Naive solution

1. `def fib(n):`
2.   `if n <= 2:`
3.     `f = 1`
4.   `else:`
5.     `f = fib(n-1) + fib(n-2)`
6.   `return f`

- Does the algorithm work?
  - Yes!
- Is it a good algorithm?
  - No!
  - Exponential time!!!
Fibonacci sequence
Naive solution

1. `def fib(n):
2.     if n <= 2:
3.         f = 1
4.     else:
5.         f = fib(n-1) + fib(n-2)
6.     return f

\[ T(n) = T(n - 1) + T(n - 2) + O(1) \]
Fibonacci sequence
Naive solution

```python
1. def fib(n):
2.     if n <= 2:
3.         f = 1
4.     else:
5.         f = fib(n-1) + fib(n-2)
6.     return f
```

\[ T(n) = T(n-1) + T(n-2) + O(1) \geq F_n \approx \varphi^n \]
Fibonacci sequence
Naive solution

1. `def fib(n):`
2.   `if n <= 2:`
3.     `f = 1`
4.   `else:`
5.     `f = fib(n-1) + fib(n-2)`
6.   `return f`

\[ T(n) = T(n-1) + T(n-2) + O(1) \geq F_n \approx \varphi^n \]
\[ \geq 2T(n-2) + O(1) \]
\[ \geq 2^{n/2} \]
fn(7) starts running
Time ≈ # calls ≈ nodes
Time ≈ # calls ≈ nodes

Space ≈ size of the longest path (root, leaf)
Fibonacci sequence
Naive solution

- We make \( n \) calls
- Calls are stored in the activation stack
Fibonacci sequence
Naive solution

1. fib(n):
2. if n <= 2:
3. f = 1
4. else:
5. f = fib(n-1) + fib(n-2)
6. return f
Fibonacci sequence
Naive solution

Time $O(2^{n/2})$

$\text{fib}(50) \approx 2^{50} \text{ steps}$

$1.12\text{e}+15 = 1.125.899.906.842.624$

```
1. fib(n):
2.   if n <= 2:
3.     f = 1
4. else:
5.     f = fib(n-1) + f(n-2)
6.   return f
```
Dynamic programming

- Features
- Overlapping problems (✅)
- Principle of optimality (??)
The principle of optimality

▷ Optimal substructure:
  ○ "A problem has optimal substructure if the optimal solution can be built from optimal solutions to its subproblems."

The principle of optimality

▷ Optimal substructure:
  ○ "A problem has optimal substructure if the optimal solution can be built from optimal solutions to its subproblems."

▷ In other words:
  ○ We can solve bigger problems using smaller instance solutions of the same problem!

\[ F_{n-2} \rightarrow F_{n-1} \rightarrow F_n \]

The principle of optimality

» Dependence on subproblems
  ○ Must form DAG (Directed Acyclic Graph)
  ○ If it has cycles, the PD algorithm can execute infinitely

Dynamic programming

- Features
- Overlapping problems (✅)
- Principle of optimality (✅)
- The dependencies of the subproblems must be acyclic (DAG!)
Dynamic programming

By using smartly, one can reduce "exponential" problems to polynomials.

Prob. must have 2 characteristics

- Overlapping problems
- Principle of optimality

Fibonacci sequence Problem

$F_n = F_{n-1} + F_{n-2}$
Memoization
A dynamic programming technique

- Remember & reuse previously computed problem solutions
Memoization
A dynamic programming technique

- Remember & reuse previously computed problem solutions
  - Maintains a "dictionary"
  - Subproblems → solutions

```plaintext
memo {
  Subp_1: val_1,
  Subp_2: val_2,
  ... : ...
  Subp_n: val_n
}
```
Memoization
A dynamic programming technique

▷ Remember & reuse previously computed problem solutions
  ○ Maintains a "dictionary"
  ○ Subproblems → solutions

▷ Recursive calls either:
  ○ Return a stored solution or
  ○ Compute and store a solution

```
memo {
    Subp_1: val_1,  
    Subp_2: val_2,
    ...  : ...  
    Subp_n: val_n,  
}
```
Fibonacci sequence
Solution using Memoization

```python
1. memo = {}
2. def fib(n):
3.     if n in memo: return memo[n]
4.     if n <= 2:  
5.         f = 1
6.     else:     
7.         f = fib(n-1) + fib(n-2)
8.     memo[n] = f
9.     return f
```
Fibonacci sequence
Solution using Memoization

1. memo = {}
2. def fib(n):
3.     if n in memo: return memo[n]
4.     if n <= 2:
5.         f = 1
6.     else:
7.         f = fib(n-1) + fib(n-2)
8.     memo[n] = f
9.     return f
Fibonacci sequence
Solution using Memoization

1. memo = {}
2. def fib(n):
3.     if n in memo: return memo[n]
4.     if n <= 2:
5.         f = 1
6.     else:
7.         f = fib(n-1) + fib(n-2)
8.     memo[n] = f
9.     return f
Fibonacci sequence
Solution using Memoization

▷ Does $\text{fib}(k)$ once for each $k$

▷ Runtime $O(n)$
  ○ Only $n$ no 'memorized' calls
  ○ $O(1)$ time per call
    ■ Ignore recursion

```python
1. memo = {}
2. def fib(n):
3.   if n in memo: return memo[n]
4.   if n <= 2:
5.     f = 1
6.   else:
7.     f = fib(n-1) + fib(n-2)
8.   memo[n] = f
9.   return f
```
Memoization
A dynamic programming technique

- The cost to compute each solution is paid only once
Memoization
A dynamic programming technique

▷ The cost to compute each solution is paid only once
▷ The cost of DP with memoization:

\[ Time \leq \sum_{\text{subproblems}} \text{Nonrecursive work} \]
Memoization
A dynamic programming technique

▷ The cost to compute each solution is paid only once
▷ The cost of DP with memoization:

\[ \text{Time} \leq \sum_{\text{subproblems}} \text{Non-recursive work} \leq \#\text{subproblems} \times \text{non-recursive work} \]

\[ O(1) \]
Memoization
A dynamic programming technique

- The cost to compute each solution is paid only once
- The cost of DP with memoization:

\[
Time \leq \sum_{\text{subproblems}} \text{Non-recursive work} \leq \#\text{subproblems} \times \text{non-recursive work} \leq O(n) \leq O(1)
\]
Context
Dynamic programming

- Second perspective on PD:
  - DP ≈ Recursion + "recycling"
Dynamic programming

Second perspective on PD:
- DP $\approx$ Recursion + "reuse"
  - Memoization ("remind") & reuse solutions to subproblems that help solve the original problem
Bottom-Up
ANOTHER dynamic programming technique

```python
def fib_botton_up(n):
    memo[0] = memo[1] = 1
    for i in range(2,n+1):
        memo[i] = memo[i-1] + memo[i-2]
    return memo[n]
```

$\begin{array}{c|c|c}
F_1 & F_2 \\
1 & 1 \\
\end{array}$
Bottom-Up
ANOTHER dynamic programming technique

1. `def fib_button_up(n):
2.     memo[0] = memo[1] = 1
3.     for i in range(2,n+1)
4.         memo[i] = memo[i-1] + memo[i-2]
5.     return memo[n]`
Bottom-Up
ANOTHER dynamic programming technique

1. `def fib_button_up(n):
2.     memo[0] = memo[1] = 1
3.     for i in range(2, n+1):
4.         memo[i] = memo[i-1] + memo[i-2]
5.     return memo[n]`
Bottom-Up
ANOTHER dynamic programming technique

```python
1. def fib_button_up(n):
2.    memo[0] = memo[1] = 1
3.    for i in range(2, n+1):
4.        memo[i] = memo[i-1] + memo[i-2]
5.    return memo[n]
```
Bottom-Up
ANOTHER dynamic programming technique

1. **def** fib_button_up(n):
2.    memo[0] = memo[1] = 1
3.    **for** i in range(2, n+1):
4.        memo[i] = memo[i-1] + memo[i-2]
5.    **return** memo[n]
Bottom-Up ANOTHER dynamic programming technique

Does the same computation as the memoized version

1. `def` `fib_button_up(n):`
2. `memo[0] = memo[1] = 1`
3. `for` `i` `in` `range(2, n+1)`
4. `memo[i] = memo[i-1] + memo[i-2]`
5. `return` `memo[n]`
Does the same computation as the memoized version

Topological ordering of subproblem dependencies (form a DAG!)

```
def fib_button_up(n):
    memo[0] = memo[1] = 1
    for i in range(2,n+1):
        memo[i] = memo[i-1] + memo[i-2]
    return memo[n]
```
Bottom-Up
ANOTHER dynamic programming technique

1. def fib_button_up(n):
2.    memo[0] = memo[1] = 1
3.   for i in range(2,n+1)
4.      memo[i] = memo[i-1] + memo[i-2]
5. return memo[n]

▷ In practice it is faster
  ○ There is no recursion
▷ The analysis is more obvious
  \[ O(n) \]
1. `def fib_button_up(n):
2.    memo[0] = memo[1] = 1
3.    for i in range(2, n+1)
4.       memo[i] = memo[i-1] + memo[i-2]
5.    return memo[n]

- In practice it is faster
  - There is no recursion
- The analysis is more obvious
- Can save space
  - We can remember only the last 2 fibs
    - Space $O(1)$
Bottom-Up
ANOTHER dynamic programming technique

- In practice it is faster
  - There is no recursion
- The analysis is more obvious
- Can save space
  - We can remember only the last 2 fibs
    - Space O(1)

```python
1. def fib_button_up(n):
2.     memo[0] = memo[1] = 1
3. for i in range(2, n+1):
4.     memo[i] = memo[i-1] + memo[i-2]
5. return memo[n]
```

There is an implementation of the seq. Time cost Fibonacci O(lg n) via a different technique!

https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/
Generic algorithms
Top-Down and Bottom-Up

1. memo = {}
2. def fib(n):
3.   if n in memo: return memo[n]
4.   if n <= 2:
5.     f = 1
6.   else:
7.     f = fib(n-1) + fib(n-2)
8.   memo[n] = f
9.   return f
Generic algorithms
Top-Down and Bottom-Up

1. `def fib_button_up(n):
   3.   for i in range(2,n+1)
   4.     memo[i] = memo[i-1] + memo[i-2]
   5.   return memo[n]

1. `def f(subprob):
   2.   Base case
   3.   for subprob:
   4.     memo[subprob] = REC relation.
   5.   original return
   6. 
Comparison between PD techniques: memoization (top-down) and tabulation (bottom-up)

<table>
<thead>
<tr>
<th></th>
<th>Tabulation (bottom-up)</th>
<th>Memoization (Top-Down)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speed</strong></td>
<td>Fast. Directly accesses dependent solutions directly from the table</td>
<td>Slow. Due to multiple recursive calls and returns</td>
</tr>
<tr>
<td><strong>Solution for subprob.</strong></td>
<td>If all subproblems must be solved at least once, DP using Bottom-up usually performs better than top-down DP</td>
<td>If not all subproblems in the subproblem space need to be solved, the solution using memoization has the advantage of solving only the necessary subproblems</td>
</tr>
<tr>
<td><strong>Memo filling</strong></td>
<td>Starts from the first entry. The other entries are filled in one by one.</td>
<td>The table is populated on demand, that is, not all entries are necessarily populated.</td>
</tr>
<tr>
<td><strong>Code</strong></td>
<td>It can become complex when you have multiple conditions</td>
<td>Typically less complicated and drawn directly from recurrence.</td>
</tr>
</tbody>
</table>
Exercises
Exercise (1)

Run the following codes to find the Fibonacci of some numbers

```python
def fibonacci_of(n):
    if n in {0, 1}:
        return n
    return fibonacci_of(n - 1) + fibonacci_of(n - 2)

print(fibonacci_of(50))
```

```python
def fibonacci_of(n):
    memo = {0: 0, 1: 1}
    if n in memo:
        return memo[n]
    memo[n] = fibonacci_of(n - 1) + fibonacci_of(n - 2)
    return memo[n]

print(fibonacci_of(50))
```
Exercise (2)

Based on the previous exercise, answer:
- Compare the runtime of the two approaches for $n = \{30, 40, 50\}$.
- Justify the time difference of both the versions.
Exercise (2)

Based on the previous exercise, answer:

- Compare the runtime of the two approaches for $n = \{30, 40, 50\}$.
- Justify the time difference of both the versions.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Fib - Recursive</th>
<th>Fib - Memoized</th>
</tr>
</thead>
<tbody>
<tr>
<td>This laptop</td>
<td>0.21s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>273.57s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>Stopped 1h+</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
</tbody>
</table>
Other Problems

▷ Calculate the fiftieth (50) Fibonacci sequence number
▷ Count the # in different ways to move in a 6x9 grid
▷ Given a set of coins, how can we make 27 cents on the fewest coins?
▷ Find the length of the longest subsequence of a given sequence such that all elements of the subsequence are ordered in ascending order
Access to information

- Reference books:
  - Algorithms Theory and Practice [CLRS]
  - Algorithm Design [Jon Keiberg, Eva Tardos]
  - Algorithms [Robert Sedgewick]
    - https://algs4.cs.princeton.edu/home/