# Lecture 16: Dynamic Programming

Agenda:

- Dynamic programming
  - concepts
  - example

Reading:

- Textbook pages 323 – 324

# Dynamic programming introduction:

- An algorithm design technique
- Usually for optimization problems
- Typically like divide-and-conquer uses solutions to subproblems to solve the problem, BUT
- Key idea:
  - *Avoids re-computation*
  - of repeated subproblems by storing subproblem answers in tables/arrays
- 1st example problem — Fibonacci numbers

$$f(n) = \begin{cases} n, & \text{when } n = 0, 1 \\ f(n-1) + f(n-2), & \text{when } n \geq 2 \end{cases}$$

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $f(n)$ | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 |

Question: how do we compute $f(n)$?

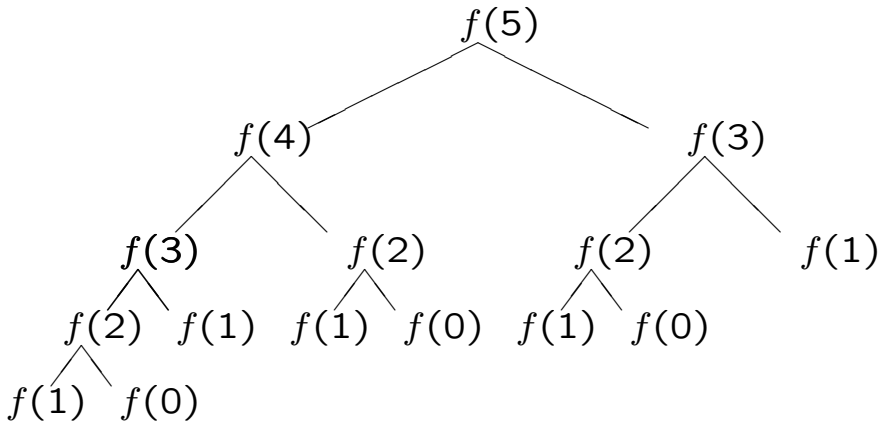
# 1st Naive Fibonacci implementation – recursion

- Pseudocode:

```
procedure f(n)

if n < 2 then
    return n
else
    return f(n − 1) + f(n − 2)
```

- Recursion tree:




- Notice that there are a lot of repeated function calls

- Running time recurrence

$$T(n) = \begin{cases} c_1, & \text{when } n = 0, 1 \\ c_2 + T(n-1) + T(n-2), & \text{when } n \geq 2 \end{cases}$$

- Conclusion: $T(n) > f(n) \longrightarrow T(n) \in \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$

- Problem with the 1st implementation — repeated function calls

- Key idea:

  Do the computation in a bottom up manner, and
  Store the compute values for future use.

  - Define array $F[0..n]$ where $F[i]$ is going to store $f(i)$.
  - Fill in the values of $F[0]$ and $F[1]$ from the definition (initialization)
  - Start computing $F[i]$, from $i = 2$ onward, using the recurrence $F[i] \longleftarrow F[i-1] + F[i-2]$.
  - Each time, we want to compute $F[i]$, $F[i-1]$ and $F[i-2]$ are *already computed!* (bottom up).

## 2nd Fibonacci implementation — dynamic programming

- Pseudocode:

```
procedure Dynfib(n)

F[0] ← 0
F[1] ← 1
for j ← 2 to n do
    F[j] ← F[j − 1] + F[j − 2]
return F[n]
```

- Running time

$$T(n) \in \Theta(n)$$

## General steps in designing a dynamic programming solution:

- Step 1: Describe an array of values that you want to compute.

  Do not say how you compute the array, but define what you store in the array. (e.g. array $F[0..n]$ and $F[i]$ is going to hold $f(i)$).

- Step 2: Give a recurrence relating some values in the array to other values in the array.

  The basis of your recurrence should specify how to initialize the array (e.g. $F[0] = 0$, $F[1] = 1$, and $F[i] = F[i-1] + F[i-2]$).

- Step 3: Give a high level program to compute the entries of the array using the recurrence above.

- Step 4: State how to extract the solution from the array (e.g. return $F[n]$).

## 2nd example problem — Knapsack

- We have a knapsack with capacity $C$
- $n$ items with weights $w_1, \ldots, w_n \in \mathbb{N}$ and values $v_1, \ldots, v_n \in \mathbb{N}$.
- Want to fill the knapsack with items to maximize the value without exceeding its capacity.
- Formally, for each $S \subseteq \{1, \ldots, n\}$ let $K(S) = \sum_{i \in S} w_i$.
- Find $M = \max_{S \subseteq \{1,\ldots,n\}} \{K(S) | K(S) \leq C\}$.
- Example: $w_1 = 10$, $w_2 = 10$, $w_3 = 15$, $v_1 = 20$, $v_2 = 19$, $v_3 = 40$, $C = 20$. Answer: pick item 3 only.

## 1st (naive) solution: try all possible subsets of items and select the best.

- Consider each set $S$ of all $2^n$ possible subsets of $\{1, \ldots, n\}$.
- Compute the weight and value of set $S$.
- Find the set with maximum value among those that have $K(S) \leq C$.
- Running time: at least $\Omega(2^n)$ subsets to consider!!

## 2nd solution: use Dynamic programming.

- Step 1: Define array $A[i, D]$, $0 \leq i \leq n$ and $0 \leq D \leq C$ where $A[i, D]$ is the value of best possible knapsack of weight at most $D$ using only items from 1 to $i$. Final sol. value: $A[n, C]$.
- Step 2: How to compute $A[i, D]$?
  - If $i = 0$ or $D = 0$ then trivially $A[i, D] = 0$.
  - Else, consider item $i$:
    - If we do not choose item $i$: knapsack must be packed optimally with items from $1 \ldots (i-1)$.
    - If we choose item $i$ (assuming $D \geq w_i$): rest of $D - w_i$ remaing cap. must be packed with items $1 \ldots (i-1)$.
    - So $A[i, D] = \max \begin{cases} A[i-1, D] \\ \text{(if } D \geq w_i\text{)} \ v_i + A[i-1, D-w_i] \end{cases}$
- Step 3:

```
procedure Knapsack

for i ← 1 to n do
    A[i, 0] ← 0
for D ← 0 to C do
    A[0, D] ← 0
for i ← 1 to n do
    for D ← 1 to C do
        A[i, D] ← A[i − 1, D]
        if D ≥ w_i and A[i, D] < A[i − 1, D − w_i] + v_i then
            A[i, D] ← A[i − 1, D − w_i] + v_i
return A[n, C]
```

- Step 4: How to find the set of items of the optimal packing?
- Running time?

Have you understood the lecture contents?

| well | ok | not-at-all | topic |
|---|---|---|---|
| ☐ | ☐ | ☐ | deriving recurrence |
| ☐ | ☐ | ☐ | avoiding re-computation |
| ☐ | ☐ | ☐ | bottom-up — dynamic programming |