CSCI3160 Design and Analysis of Algorithms (2025 Fall)

Dynamic Programming 1: Pitfall of Recursion

Instructor: Xiao Liang[1]

Department of Computer Science and Engineering
The Chinese University of Hong Kong

[1] These slides are primarily based on materials prepared by Prof. Yufei Tao (please refer to Prof. Tao's version from 2024 Fall for the original content). Some modifications have been made to better align with this year's teaching progress, incorporating student feedback, in-class interactions, and my own teaching style and research perspective.

Problem: Let A be an array of n positive integers.

Consider function

$$f(k) = \begin{cases} 0 & \text{if } k = 0 \\ \max_{i=1}^{k}(A[i] + f(k-i)) & \text{if } 1 \le k \le n \end{cases}$$

Goal: Compute $f(n)$.

Example: Consider the following example with $n = 4$ and the array $A = [1, 5, 8, 9]$:

i	1	2	3	4
$A[i]$	1	5	8	9
$f(i)$	1	5	8	10

Pitfall of Recursion

Consider the following recursive algorithm for computing $f(k)$.

f(k)

if $k = 0$ **then return** 0
$\quad$ $ans \leftarrow -\infty$
for $i \leftarrow 1$ to k **do**
$\qquad$ $tmp \leftarrow A[i] + \mathbf{f}(k - i)$
$\qquad$ **if** $tmp > ans$ **then** $ans \leftarrow tmp$
return ans

Computing $f(n)$ with the above algorithm incurs running time $\Omega(2^n)$ (left as a regular exercise).

Pitfall of Recursion

f(k)

if $k = 0$ **then return** 0
$ans \leftarrow -\infty$
for $i \leftarrow 1$ to k **do**
    $tmp \leftarrow A[i] + \mathbf{f}(k - i)$
    if $tmp > ans$ **then** $ans \leftarrow tmp$
return ans

Why is the algorithm so slow?

> **Answer:** It computes $f(x)$ for the same x repeatedly!

How many times do we need to call **f**(0) in computing $f(1)$, $f(2)$, ..., and $f(6)$, respectively?

Pitfall of recursion:

A recursive algorithm does considerable redundant work if the **same** subproblem is encountered over and over again.

Idea:

We can hope for a better solution if the following conditions are satisfied:

- We can store the solutions to subproblems for later reuse.
- These subproblems appear in a specific order that allows us to use the solution of an earlier subproblem to solve a later one.
- We can efficiently determine (i.e., compute) that specific order.

These conditions constitute the core principle of dynamic programming! I.e.,

> Resolve subproblems according to a certain **order**. Remember the output of every subproblem to avoid re-computation.

Example Illustrating the Idea

Problem: Let A be an array of n positive integers. Compute $f(n)$.

$$f(k) = \begin{cases} 0 & \text{if } k = 0 \\ \max_{i=1}^{k}(A[i] + f(k-i)) & \text{if } 1 \leq k \leq n \end{cases}$$

Order of subproblems: $f(1)$, ..., $f(n)$.

Solve subproblem $f(1)$: $O(1)$ time
Solve subproblem $f(2)$: $O(2)$ time, **given** $f(1)$.
$\vdots$
Solve subproblem $f(n)$: $O(n)$ time, **given** $f(1)$, ..., $f(n-1)$.

In total: $O(n^2)$ time.

Pseudocode of our algorithm:

dyn-prog

```
initialize an array ans of size n
define special value ans[0] ← 0
for k ← 1 to n do        /* assuming f(0), f(1), ..., f(k − 1) ready, compute f(k) */
    ans[k] ← −∞
    for i ← 1 to k do
        tmp ← A[i] + ans[k − i]
        if tmp > ans[k] then ans[k] ← tmp
```

Time complexity: $O(n^2)$.

Why is it called *Dynamic Programming*?

The name is misleading!

Dynamic Programming has little to do with "programming" in the modern sense of writing code.

- The term was coined by **Richard Bellman** in the 1950s.
- At the time, "programming" referred to *planning or scheduling*, like in "linear programming."
- "Dynamic" emphasized that the method solves problems by breaking them down over time or stages.

So don't let the name confuse you—it's about solving problems efficiently by reusing solutions to subproblems.