Dynamic Programming 1: Pitfall of Recursion

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

Today, we will start a series of lectures on **dynamic programming**, which is a technique for accelerating recursive algorithms.

> **Remark:** Despite the word "programming", the technique has nothing to do with programming languages.

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

Consider function

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

Goal: Compute $f(n)$.

Example: Consider the following array A:

i	1	2	3	4
$A[i]$	1	5	8	9

Then, $f(1) = 1, f(2) = 5, f(3) = 8$, and $f(4) = 10$.

Pitfall of Recursion

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

f(k)

```
if k = 0 then return 0
ans ← −∞
for i ← 1 to k do
    tmp ← A[i] + f(k − i)
    if tmp > ans then ans ← 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)
1. **if** $k = 0$ **then return** 0
2. $ans \leftarrow -\infty$
3. **for** $i \leftarrow 1$ to k **do**
4. $\quad tmp \leftarrow A[i] +$ **f**$(k - i)$
5. $\quad$ **if** $tmp > ans$ **then** $ans \leftarrow tmp$
6. **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.

Antidote: dynamic programming.

Principle of dynamic programming

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

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

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

Goal: Compute $f(n)$.

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

Resolve subproblem $f(1)$: $O(1)$ time
Resolve subproblem $f(2)$: $O(2)$ time, **given** $f(1)$.
...
Resolve subproblem $f(k)$: $O(k)$ time, **given** $f(1)$, ..., $f(k-1)$.
...
Resolve 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: $O(n^2)$.