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 \( A \):

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A[i] )</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Then, \( f(1) = 1 \), \( f(2) = 5 \), \( f(3) = 8 \), and \( f(4) = 10 \).
Consider the following recursive algorithm for computing $f(k)$.

\[
\begin{align*}
f(k) & \\
1. & \textbf{if } k = 0 \textbf{ then return } 0 \\
2. & \quad \text{ans } \leftarrow -\infty \\
3. & \quad \textbf{for } i \leftarrow 1 \textbf{ to } k \textbf{ do} \\
4. & \quad \quad \text{tmp } \leftarrow A[i] + f(k - i) \\
5. & \quad \quad \textbf{if tmp } > \text{ans } \textbf{ then ans } \leftarrow \text{tmp} \\
6. & \quad \textbf{return ans}
\end{align*}
\]

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

\[ f(k) \]
1. \( \text{if } k = 0 \text{ then return } 0 \)
2. \( \text{ans } \leftarrow -\infty \)
3. \( \text{for } i \leftarrow 1 \text{ to } k \text{ do} \)
4. \( \quad \text{tmp } \leftarrow A[i] + f(k - i) \)
5. \( \quad \text{if } \text{tmp} > \text{ans} \text{ then } \text{ans } \leftarrow \text{tmp} \)
6. \( \text{return ans} \)

Why is the algorithm so slow?

**Answer:** It re-computes \( f(x) \) for the same \( x \) repeatedly!

How many times do we need to call \( f(2) \) in computing \( f(3), f(4), f(5), \) 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.

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

**Goal:** Compute $f(n)$.

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

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 $f(1), \ldots, f(k - 1)$.

... 
Resolve subproblem $f(n)$: $O(n)$ time, given $f(1), \ldots, f(n - 1)$.

In total: $O(n^2)$ time.
Pseudocode of our algorithm:

**dyn-prog**
1. initialize an array $ans$ of size $n$
2. define special value $ans[0] \leftarrow 0$
3. for $k \leftarrow 1$ to $n$ do
   /* assuming $f(0), f(1), \ldots, f(k - 1)$ ready, compute $f(k)$ */
4. $ans[k] \leftarrow -\infty$
5. for $i \leftarrow 1$ to $k$ do
6. $tmp \leftarrow A[i] + ans[k - i]$
7. if $tmp > ans[k]$ then $ans[k] \leftarrow tmp$

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