Recursion, memoization, dynamic programming

L3 Algorithmics and Programming

Pierre Senellart

ÉCOLE NORMALE SUPÉRIEURE | PSL RESEARCH UNIVERSITY PARIS

11 October 2018

Outline

When should one use recursion?

A problem P can be solved by recursion when:

- It can be parameterized by one or several integers $P(n_1 \dots n_k)$
- The solution of $P(n_1 \dots n_k)$ can be obtained from the solution of $P(n_1^{(1)} \dots n_k^{(1)}) \dots P(n_1^{(\ell)} \dots n_k^{(\ell)})$ for some $\ell \geqslant 1$ with, for all $1 \leqslant i \leqslant \ell$, $n_j^{(i)} \leqslant n_j$ for $1 \leqslant j \leqslant k$, one of these inequalities being strict $(\forall i, \exists j, n_j^{(i)} < n_j)$: recursive case
- $P(0 \dots 0)$ (or $P(n_1 \dots n_k)$ with n_i “small” depending on the case) easy to compute: base case

Example 1: Factorial

- $P(n) = n!$
- $k = 1$, $\ell = 1$
- $P(n) = n \times P(n-1)$ for $n \geqslant 1$
- $P(0) = 1$

Example 2: Fibonacci

- $P(n)$ nth Fibonacci number
- $k = 1$, $\ell = 2$
- $P(n) = P(n-1) + P(n-2)$ for $n \geqslant 1$
- $P(0) = 1$, $P(1) = 1$

Example 3: Levenshtein edit distance

$d(s, s')$ edit distance (minimal number of characters to add, remove, modify) to go from s to s'.

- $P(n_1, n_2)$ edit distance between the prefix of length n_1 of s and the prefix of length n_2 of s'
- $k = 2$, $\ell = 3$
- $P(n_1, n_2) = \min\left(P(n_1 - 1, n_2) + 1, P(n_1, n_2 - 1) + 1, P(n_1 - 1, n_2 - 1) + \mathbb{I}_{s_{n_1} \neq s'_{n_2}}\right)$ for $n_1 \geqslant 1$, $n_2 \geqslant 1$
- $P(n_1, 0) = n_1$ for $n_1 \geqslant 0$; $P(0, n_2) = n_2$ for $n_2 \geqslant 0$.

$\mathbb{I}_b$ is 1 if b is true, 0 otherwise

Implementation

- Recursive function

```
function P(n_1, ..., n_k);
if base case then
    ...;
    return ...;
else
    // recursive case
    ...;
    // ℓ calls to P
    return ...;
end
```

- One says recursion is terminal if $\ell = 1$ and the call to P in the last instruction of the recursive case (**return** $P(n'_1, \ldots, n'_k)$)
- NB: On the example above, the **else** can be omitted.

Complexity

- Upper bounds (sometimes better than that, e.g., if recursive call is on $\frac{n}{2}$ and not on $n-1$):
  - If $\ell = 1$: $O(n_1 + \cdots + n_k)$: often acceptable
  - If $\ell > 1$: $O(\ell^{n_1 + \cdots + n_k})$: often unreasonable
- Be wary of memory used on the stack: $O(\sum_{i=1}^{k} n_i)$ (usually, memory allocated to the stack varies from hundreds of kilobytes to a few megabytes).
- When recursion is terminal, the compiler may be able to eliminate recursion, and therefore stack space requirements.

Outline

Idea (1/3)

Cache: global object (or static, or class member) that remembers results from function P from one call to another

Idea (2/3)

function $P(n_1, \ldots, n_k)$;
if $M(n_1, \ldots, n_k)$ *is defined* **then**
    return $M(n_1, \ldots, n_k)$;
end
if *base case* **then**
    $\ldots$;
    $r \leftarrow \ldots$;
else
    `// recursive case`
    $\ldots$;
    `//` ℓ `calls to` P
    $r \leftarrow \ldots$;
end
$M(n_1, \ldots, n_k) \leftarrow r$;
return r;

Idea (3/3)

- Terminal recursion is impossible!
- Data structure: multi-dimensional array or associative array (map) implemented as a balanced search tree or as a hash table (cf. upcoming lecture)
- Use a classic array when parameters are integers with contiguous values, use an associative array otherwise
- Array can generally be statically allocated when one has an a priori upper bound on parameter sizes

Complexity

- $O(n_1 \times \cdots \times n_k \times \ell)$ operations
- $O(n_1 \times \cdots \times n_k)$ memory used on the heap (space used depends on data structure)
- $O(n_1 + \cdots + n_k)$ memory used on the stack
- No useless computation!

Outline

Idea

- Iterative computation
- One builds a matrix (multi-dimensional array) $n_1 \times \cdots \times n_k$ that is iteratively filled, bottom up

function $P(n_1, \ldots, n_k)$;
$M \leftarrow \text{matrix}(n_1, \ldots, n_k)$;
`// fill` M `with base cases`
for $i_1 \leftarrow 1$ **to** n_1 **do**
  $\ldots$;
  for $i_k \leftarrow 1$ **to** n_k **do**
    $\ldots$;
    $M(i_1, \ldots, i_k) \leftarrow \ldots$;
    `// use precomputed values`
  end
end
return $M(n_1 \ldots n_k)$;

Complexity

- $\Theta(n_1 \times \cdots \times n_k \times \ell)$ operations
- $\Theta(n_1 \times \cdots \times n_k)$ memory used on the heap
- $O(1)$ memory used on the stack
- Sometimes too many operations!

Memoization vs dynamic programming

- The two techniques roughly amount to the same thing
- Pro of memoization: only necessary computations are made (the matrix is not fully filled)
- Con of memoization: recursive calls have a small overhead (wrt imperative style) and use the call stack (limited in size)
- The behavior of dynamic programming can be simulated with memoization
- For some complex cases, one can simulate recursive calls with a stack maintained within the heap