COMP122/20 - Data Structures and Algorithms

10 Tail Recursion and Loops

Instructor : Ke Wei 〚柯韋〛

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/20/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

February 21, 2020

Outline

Textbook §4.2, 4.4 – 4.6.

Computing Integer Square Roots

- Given an integer $x \geqslant 0$, the integer square root $int\_sqrt(x)$ returns an integer r such that

$$r^2 \leqslant x \leqslant (r+1)^2 - 1.$$

- Obviously, $int\_sqrt(x) = \lfloor \sqrt{x} \rfloor$.
- Since this is a problem related to multiplication, we try to reduce the problem to solving the integer square root of $y = \lfloor \frac{x}{4} \rfloor$. Thus, we have $4y \leqslant x \leqslant 4y+3$.
- Let $s = int\_sqrt(y)$, that is, $s^2 \leqslant y \leqslant (s+1)^2 - 1$, we have

$$(2s)^2 = 4s^2 \leqslant 4y \leqslant x \leqslant 4y+3 \leqslant 4(s+1)^2 - 4 + 3 = (2s+2)^2 - 1.$$

- Therefore, the integer square root of x can be computed as

$$int\_sqrt(x) = \begin{cases} 0 & \text{if } x = 0, \\ 2s+1 & \text{otherwise, if } (2s+1)^2 \leqslant x, \\ 2s & \text{otherwise.} \end{cases}$$

Computing Integer Square Roots — Code

```python
def int_sqrt(x):
    if x == 0:
        return 0
    else:
        r = 2*int_sqrt(x//4)+1
        return r if r*r <= x else r-1
```

The recursive calls for computing the integer square roots of 100, 900 and 4000 are expanded respectively below.

$$\begin{array}{ccccccccccc} 100 & \Rightarrow & 25 & \Rightarrow & 6 & \Rightarrow & 1 & \Rightarrow & 0 \\ 11,\boxed{10} & \Leftarrow & \boxed{5},4 & \Leftarrow & 3,\boxed{2} & \Leftarrow & \boxed{1},0 & \Leftarrow & 0 \end{array} \qquad \begin{array}{ccccccccccc} 900 & \Rightarrow & 225 & \Rightarrow & 56 & \Rightarrow & 14 & \Rightarrow & 3 & \Rightarrow & 0 \\ 31,\boxed{30} & \Leftarrow & \boxed{15},14 & \Leftarrow & \boxed{7},6 & \Leftarrow & \boxed{3},2 & \Leftarrow & \boxed{1},0 & \Leftarrow & 0 \end{array}$$

$$\begin{array}{ccccccccccccc} 4000 & \Rightarrow & 1000 & \Rightarrow & 250 & \Rightarrow & 62 & \Rightarrow & 15 & \Rightarrow & 3 & \Rightarrow & 0 \\ \boxed{63},62 & \Leftarrow & \boxed{31},30 & \Leftarrow & \boxed{15},14 & \Leftarrow & \boxed{7},6 & \Leftarrow & \boxed{3},2 & \Leftarrow & \boxed{1},0 & \Leftarrow & 0 \end{array}$$

Permutation of List Elements

- How to permute elements $e_0, e_1, \dots, e_{n-1}$ in the first r positions of list a, that is, in $a[0:r]$?
  - Place e_0 in $a[0]$, and permute the remaining elements $e_1, e_2, \dots, e_{n-1}$ in $a[1:r]$;
  - Place e_1 in $a[0]$, and permute the remaining elements $e_0, e_2, \dots, e_{n-1}$ in $a[1:r]$;

    $\vdots$
  - Place e_{n-1} in $a[0]$, and permute the remaining elements $e_0, e_1, \dots, e_{n-2}$ in $a[1:r]$.
- How many permutations all together?

$$P(n,r) = \begin{cases} 1 & \text{if } n \geqslant r = 0, \\ n \times P(n-1, r-1) & \text{if } n \geqslant r \geqslant 1. \end{cases}$$

- How do we store the "remaining" elements? Initially, we place e_i in $a[i]$, if we need to place e_i in $a[0]$, we just swap $a[0]$ and $a[i]$:

$$\boxed{e_0}, e_1, \dots, \boxed{e_i}, e_{i+1}, \dots, e_{n-1} \Longrightarrow \boxed{e_i}, \underbrace{e_1, \dots, \boxed{e_0}, e_{i+1}, \dots, e_{n-1}}_{\text{the remaining elements, starting from } a[1]}.$$

Permutation of List Elements — Code

When we reduce the permutation of all the elements to the permutation of the remaining elements, the starting location changes from $a[0]$ to $a[1]$. This location can further change if we keep reducing the subproblems. We must use a parameter s to specify the starting location.

```python
def permute(a, s, r):
    if s < r:
        for i in range(s, len(a)):
            a[s], a[i] = a[i], a[s]
            yield from permute(a, s+1, r)
            a[s], a[i] = a[i], a[s]   # restore a to its original state
    else:
        yield a[0:r]
```

Generating Permutations in Lexicographic Order

- The *lexicographic order* of two vectors is defined as $a_1a_2\ldots a_i\ldots a_n < b_1b_2\ldots b_i\ldots b_n$ if and only if there exists $1 \leqslant i \leqslant n$ such that $a_i < b_i$ and for all $1 \leqslant j < i$, $a_j = b_j$.
- The following adjustment will list the permutations in lexicographic order, if the input list a is ordered. Why?

```python
def permute(a, s, r):
    if s < r:
        for i in range(s, len(a)):
            a[s], a[i] = a[i], a[s]
            yield from permute(a, s+1, r)
        a[s:] = a[s+1:]+a[s:s+1]   # restore a to its original state
    else:
        yield a[0:r]
```


Some Recursive Mathematical Functions

Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \times (n-1)! & \text{if } n \geqslant 1. \end{cases}$$

Fibonacci numbers

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

$$fib(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ fib(n-1) + fib(n-2) & \text{if } n \geqslant 2. \end{cases}$$

Going upstairs: you can choose to step either one or two stairs at a time, how many ways to go up n-stairs? For example, to go up 4 stairs, you have 5 ways:

$$1 \to 1 \to 1 \to 1 \qquad 1 \to 1 \to 2 \qquad 1 \to 2 \to 1 \qquad 2 \to 1 \to 1 \qquad 2 \to 2.$$

Tail Recursion

- If we don't fix the first two items of the Fibonacci sequence, instead, we specify them as parameters a and b, then the Fibonacci numbers can be defined as $fib(n) = fib\_t(n, 0, 1)$, where

$$fib\_t(n, a, b) = \begin{cases} a & \text{if } n = 0, \\ b & \text{if } n = 1, \\ fib\_t(n-2, a+b, b+(a+b)) & \text{if } n \geqslant 2. \end{cases}$$

$fib\_t(n,0,1)$	0	1	2	3	4	5	6	7	8	9	10	11
	0	1	1	2	3	5	8	13	21	34	55	89
$fib\_t(n,1,2)$			0	1	2	3	4	5	6	7	8	9

- This is a tail recursion, where every recursive call is the last call before return.
- A tail recursion can be transformed to a loop directly.

Converting Tail Recursions to Loops

- The loop condition is the condition when you do recursive calls.
- The loop body consists of the statements under the condition.
- Each tail recursive call is replaced by an iteration step, where *the function formal paramenters* are assigned with *the actual arguments in the call*.
- The base cases are placed after the loop.

```
def fib_t(n, a, b):
    # fib_t(n,a,b) = fib_t(n−2, a+b, b+(a+b))   for n ⩾ 2.


    while n >= 2:
        n, a, b = n-2, a+b, b+(a+b)
    return a if n == 0 else b
```


Converting Tail Recursions to Loops — Factorial

- The factorial function can also be defined as a tail recursion with an additional *accumulator parameter p*:

$$fact\_t(n,p) = p \times n! \begin{cases} p & \text{if } n = 0, \\ p \times (n \times (n-1)!) = (p \times n) \times (n-1)! & \\ \qquad\qquad = fact\_t(n-1, p \times n) & \text{if } n \geqslant 1. \end{cases}$$

- We transform the function definition into the following loop:

```
def fact_t(n, p):
    while n > 0:
        p, n = p*n, n-1
    return p
```


Computing Integer Powers

- A naïve algorithm to compute integer powers of a number x would be:

$$pow\_na(x,n,p) = px^n = \begin{cases} p & \text{if } n = 0, \\ (px)x^{n-1} = pow\_na(x, n-1, px) & \text{if } n \geqslant 1. \end{cases}$$

- A faster reduction can be achieved by dividing the exponent in half:

$$pow\_sq(x,n,p) = px^n = \begin{cases} p & \text{if } n = 0, \\ p(x^2)^k = pow\_sq(x^2, k, p) & \text{if } n = 2k \geqslant 2, \\ (px)(x^2)^k = pow\_sq(x^2, k, px) & \text{if } n = 2k+1 \geqslant 1. \end{cases}$$

- We convert the tail recursion into a loop:

```
def pow_sq(x, n, p):
    while n > 0:
        x, n, p = x*x, n//2, p if n%2 == 0 else p*x
    return p
```