

# Contents

<b>1</b>	<b>Functions</b>	<b>2</b>
1.1	algorithm – basic number theoretic algorithms . . . . .	2
1.1.1	digital_method – univariate polynomial evaluation . . . .	2
1.1.2	digital_method_func – function of univariate polynomial evaluation . . . . .	2
1.1.3	rl_binary_powering – right-left powering . . . . .	3
1.1.4	lr_binary_powering – left-right powering . . . . .	3
1.1.5	window_powering – window powering . . . . .	3
1.1.6	powering_func – function of powering . . . . .	4

# Chapter 1

## Functions

### 1.1 algorithm – basic number theoretic algorithms

#### 1.1.1 digital\_method – univariate polynomial evaluation

```
digital_method(coefficients: list, val: object, add: function, mul:  
function, act: function, power: function, zero: object, one: object )  
→ object
```

Evaluate a univariate polynomial corresponding to `coefficients` at `val`.

If the polynomial corresponding to `coefficients` is of  $R$ -coefficients for some ring  $R$ , then `val` should be in an  $R$ -algebra  $D$ .

`coefficients` should be a descending ordered list of tuples  $(d, c)$ , where  $d$  is an integer which expresses the degree and  $c$  is an element of  $R$  which expresses the coefficient. All operations 'add', 'mul', 'act', 'power', 'zero', 'one' should be explicitly given, where:

'add' means addition ( $D \times D \rightarrow D$ ), 'mul' multiplication ( $D \times D \rightarrow D$ ), 'act' action of  $R$  ( $R \times D \rightarrow D$ ), 'power' powering ( $D \times \mathbf{Z} \rightarrow D$ ), 'zero' the additive unit (an constant) in  $D$  and 'one', the multiplicative unit (an constant) in  $D$ .

#### 1.1.2 digital\_method\_func – function of univariate polynomial evaluation

```
digital_method(add: function, mul: function, act: function, power:  
function, zero: object, one: object )  
→ function
```

Return a function which evaluates polynomial corresponding to 'coefficients' at 'val' from an iterator 'coefficients' and an object 'val'.

All operations 'add', 'mul', 'act', 'power', 'zero', 'one' should be inputted in

a manner similar to **digital\_method**.

### 1.1.3 rl\_binary\_powering – right-left powering

```
rl_binary_powering(element: object, index: integer, mul: function,  
square: function=None, one: object=None, )  
→ object
```

Return `element` to the `index` power by using right-left binary method.

`index` should be a non-negative integer. If `square` is None, `square` is defined by using `mul`.

### 1.1.4 lr\_binary\_powering – left-right powering

```
lr_binary_powering(element: object, index: integer, mul: function,  
square: function=None, one: object=None, )  
→ object
```

Return `element` to the `index` power by using left-right binary method.

`index` should be a non-negative integer. If `square` is None, `square` is defined by using `mul`.

### 1.1.5 window\_powering – window powering

```
window_powering(element: object, index: integer, mul: function,  
square: function=None, one: object=None, )  
→ object
```

Return `element` to the `index` power by using small-window method.

The window size is selected by average analytic optimization.

`index` should be a non-negative integer. If `square` is None, `square` is defined by using `mul`.

### 1.1.6 powering\_func – function of powering

```
powering_func(mul: function, square: function=None, one: object=None, type: integer=0 )  
→ function
```

Return a function which computes 'element' to the 'index' power from an object 'element' and an integer 'index'.

If `square` is `None`, `square` is defined by using `mul`. `type` should be an integer which means one of the following:

```
0; rl_binary_powering  
1; lr_binary_powering  
2; window_powering
```

#### Examples

```
>>> d_func = algorithm.digital_method_func(  
... lambda a,b:a+b, lambda a,b:a*b, lambda i,a:i*a, lambda a,i:a**i,  
... matrix.zeroMatrix(3,0), matrix.identityMatrix(3,1)  
... )  
>>> coefficients = [(2,1), (1,2), (0,1)] # X^2+2*X+I  
>>> A = matrix.SquareMatrix(3, [1,2,3]+[4,5,6]+[7,8,9])  
>>> d_func(coefficients, A) # A**2+2*A+I  
[33L, 40L, 48L]+[74L, 92L, 108L]+[116L, 142L, 169L]  
>>> p_func = algorithm.powering_func(lambda a,b:a*b, type=2)  
>>> p_func(A, 10) # A**10 by window method  
[132476037840L, 162775103256L, 193074168672L]+[300005963406L, 368621393481L,  
437236823556L]+[467535888972L, 574467683706L, 681399478440L]
```