

# Introduction to Algorithms and Data Structures

## Lecture 2: Inefficient vs. efficient algorithms

John Longley

School of Informatics  
University of Edinburgh

19 September 2024

## Inefficient vs. efficient algorithms

**Goal of lecture:** Introduce some examples of algorithms, illustrating the difference between 'efficient' and 'inefficient' solutions to a problem.

**Problem 1** (toy example): Given a large decimal whole number  $n$ , compute  $n \bmod 9$  (i.e. the *remainder* on dividing  $n$  by 9).

**Method A:** Do division by school method; note the remainder.

$$\begin{array}{r} 416473525377289 \\ 9 \overline{) 371458426631472248336965268087} \text{ rem } 6 \end{array}$$

This is the 'obvious' method: no cleverness involved.

## Alternative method

**Method B:** Add the digits of  $n$  to get a new number  $n'$ .

Do the same to  $n'$ ; repeat till we get down to a single digit  $d$ ,

If  $d = 9$ , answer is 0; otherwise answer is  $d$ .

E.g. for  $n = 3748261728395607$ :

$$\begin{aligned} 3+7+4+8+2+6+1+7+2+8+3+9+5+6+0+7 &= 78 \\ 7+8 &= 15 \\ 1+5 &= 6 \end{aligned}$$

Why does this work?

E.g. for a 4-digit number written  $abcd$ :

$$\begin{aligned} 1000a + 100b + 10c + d &= (999a + a) + (99b + b) + (9c + c) + d \\ &\equiv a + b + c + d \pmod{9} \end{aligned}$$

# Comparing methods A and B

## Advantages of method B:

- ▶ Faster, at least for humans (though not spectacularly so).
- ▶ More flexible: can add digits in whatever order you like.
- ▶ Can be parallelized (you add first 8 digits, I'll add last 8).

**Moral:** Using mathematical insight, improvements over the 'obvious' algorithm may be possible.

Improved algorithm may be 'non-obvious' and may need justifying.

# Modular exponentiation

**Problem 2:** Given (large) whole numbers  $a, n, m$ , compute  $a^n \bmod m$ . E.g.  $2^{10} \bmod 17 = 1024 \bmod 17 = 4$ .

Believe it or not, this problem is absolutely fundamental to modern cryptosystems (e.g. [RSA](#), as explained in DMP course).

**Method A:** Literally compute  $a^n$ , then reduce modulo  $m$ .

- ▶ If e.g.  $a = 3$ ,  $n = 123456789012345678901234$ , then  $a^n$  won't even fit in memory.
- ▶ In any case, working with very big numbers is time-consuming.

## Modular exponentiation, continued

**Method B:** Start from  $a$ .

Do  $(n - 1)$  multiplications by  $a$ , but *reduce mod  $m$  each time*.

Works because:

$$(x \times y) \bmod m = ((x \bmod m) \times (y \bmod m)) \bmod m$$

E.g. for  $2^{10} \bmod 17$ :

$$2 \times 2 = 4, \quad 4 \times 2 = 8, \quad 8 \times 2 = 16, \quad 16 \times 2 = 32 \equiv 15, \quad 15 \times 2 = 30 \equiv 13, \\ 13 \times 2 = 26 \equiv 9, \quad 9 \times 2 = 18 \equiv 1, \quad 1 \times 2 = 2, \quad 2 \times 2 = 4.$$

- ▶ Now numbers never get bigger than  $am$ .
- ▶ But still impractical if  $n = 123456789012345678901234$ .

## Fast modular exponentiation

**Method C:** Notice that it's easy to compute  $e = a^n \pmod m$  if we've already computed  $d = a^{\lfloor n/2 \rfloor} \pmod m$ :

- ▶ If  $n$  is even, take  $e = (d \times d) \pmod m$ .
- ▶ If  $n$  is odd, take  $e = (d \times d \times a) \pmod m$ .

This suggests the following **recursive algorithm**:

```
Expmod (a,n,m):           # Computes  $a^n \pmod m$ 
    if n=0 then return 1
    else
        d = Expmod (a,⌊n/2⌋,m)
        if n is even
            return (d × d) mod m
        else return (d × d × a) mod m
```

(Example of **pseudocode**: informal mix of programming constructs, math notation, and English. Useful for expressing algorithms in a readable way.)

## Example of Method C

```
Expmod (a,n,m):  
  if n=0 then return 1  
  else  
    d = Expmod (a,[n/2],m)  
    if n is even  
      return (d × d) mod m  
    else return (d × d × a) mod m
```

Imagine each evaluation of **Expmod** is done by a different 'person':

Us to A:           What's **Expmod** (2,10,17) ?  
A to B:            What's **Expmod** (2,5,17) ?  
B to C:            What's **Expmod** (2,2,17) ?  
C to D:            What's **Expmod** (2,1,17) ?  
D to E:            What's **Expmod** (2,0,17)?  
E to D:            1  
D to C:             $1 \times 1 \times 2 \pmod{17} = 2$   
C to B:             $2 \times 2 \pmod{17} = 4$   
B to A:             $4 \times 4 \times 2 \pmod{17} = 15$   
A to us:            $15 \times 15 \pmod{17} = 4.$

This is feasible even when  $a, n, m$  are large (say  $\sim 1000$  digits).

## Some Python experiments



Time in milliseconds to compute  $3^n \bmod 2n$  (on my laptop):

$n$	Method A	Method B	Method C	$\min(A,B)/C$
10	.0027	.0051	.0055	0.49
100	.0041	.0134	.0071	0.58
1000	.0092	.1229	.0076	1.21
10000	.133	2.41	.010	13.3
100000	2.81	10.46	.016	176.
1000000	101.	127.	.017	5941.
10000000	3986.	1168.	.017	68700.
100000000	149000	11150	.019	587000.
1000000000	crashed	213000	.022	9680000.
$10^{100}$	—	—	.656	—

**Key idea:** It's not just that my **Python program** for C is the best. Rather, the **algorithm itself** is vastly, fundamentally superior.

*How do we make this idea precise?*

## Digression: Primality testing

**Fermat's little theorem:** If  $n$  is prime and  $0 < a < n$ , then  $a^{n-1} \bmod n = 1$ . [Proved in DMP.]

**Application:** Let  $n$  be the following 270-digit number.

```
412023436986659543855531365332575948179811699844327982845455
626433876445565248426198098870423161841879261420247188869492
560931776375033421130982397485150944909106910269861031862704
114880866970564902903653658867433731720813104105190864254793
282601391257624033946373269391
```

My Python 'Program C' takes  $< 5$  ms to discover that  $2^{n-1} \bmod n = \dots \neq 1$ . Conclusion:  $n$  is not prime.

So what are its factors? If you knew, you'd be (slightly) famous. This is **RSA-896**, not yet cracked. (\$75,000 prize sadly withdrawn!)

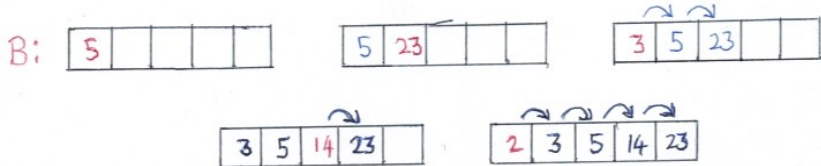
**Note:** This isn't a perfect primality test: a few non-primes (e.g. 561) masquerade as primes. But for big numbers, error probability is very small — and by refining the test, can be made even smaller (**Miller-Rabin test**).

# InsertSort

**Problem 3:** Given an array  $A$  containing  $n$  whole numbers, construct an array  $B$  containing the same  $n$  numbers in non-decreasing order.

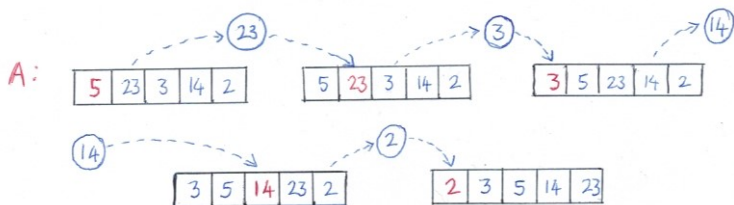


**Method A** ('obvious'): Go through elements of  $A$  one by one. Copy them into  $B$ , filling  $B$  from left to right, and inserting each element in its correct position.



## InsertSort: in-place version

Actually, don't need a separate array  $B$ : can do everything within  $A$  itself (**in-place sorting**). Just need to be able to hold one number 'in our hand' at any given time.



In pseudocode:

**InsertSort(A):**

for  $i = 1$  to  $|A|-1$     #  $|A|$  means size of  $A$

$x = A[i]$

$j = i-1$

    while  $j \geq 0$  and  $A[j] > x$

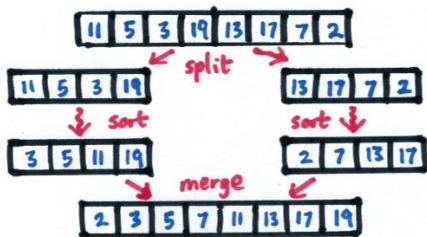
$A[j+1] = A[j]$

$j = j-1$

$A[j+1] = x$

# MergeSort

Method B (less obvious). Split A into two halves. Sort these separately, then merge the results.



## Merge (B,C):

allocate D of size  $|B| + |C|$

$i = j = 0$

for  $k = 0$  to  $|D| - 1$

if  $B[i] < C[j]$  # Convention:  $\infty$  if index out of range

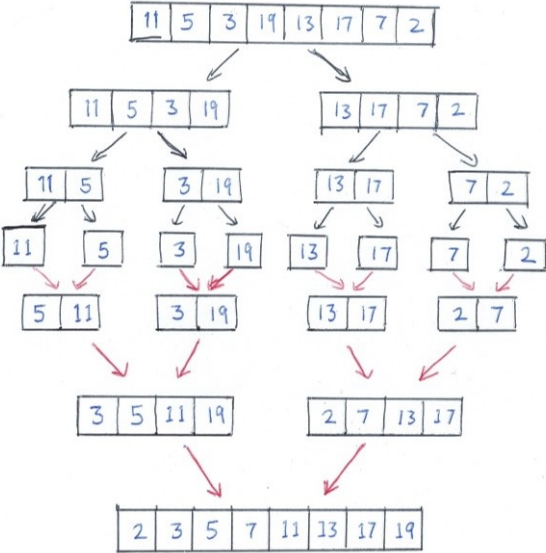
$D[k] = B[i], i = i + 1$

else

$D[k] = C[j], j = j + 1$

return D

# MergeSort: Recursive application of Merge



## MergeSort continued

A **recursive** sorting algorithm:

```
MergeSort (A,m,n):    # sorts A[m], A[m+1], ..., A[n-1]
                       # returning result in an array D of size n-m
    if n-m = 1
        return [ A(m) ]
    else
        p =  $\lfloor (m+n)/2 \rfloor$ 
        B = MergeSort (A,m,p)
        C = MergeSort (A,p,n)
        D = Merge (B,C)
        return D
```

```
MergeSortAll (A):
    return MergeSort (A,0,|A|)
```

## Python again



Time in milliseconds to sort a list of length  $n$ .  
(Entries were random whole numbers  $< n^2$ .)

$n$	InsertSort	MergeSort	Speedup
10	.023	.068	0.34
100	.97	.74	1.31
1000	69.8	7.9	8.84
10000	8210.	76.2	107.
100000	906000.	1080.	839.
1000000	–	13300.	
10000000	–	158000.	
100000000	–	2619000.	

So MergeSort seems fundamentally superior (as regards runtime).  
Again, **how can we make this precise?**

And **why** is MergeSort so much better? What's going on here?

**Will explore this next time.**

## Reading

- ▶ InsertSort: CLRS 2.1
- ▶ MergeSort: Roughgarden 1.4,1.5; CLRS 2.3
- ▶ Modular exponentiation: CLRS 31.6, second half
- ▶ [RSA challenge numbers: good Wikipedia pages.]