Introduction to Algorithms and Data Structures
Lecture 2: Inefficient vs. efficient algorithms

John Longley

School of Informatics
University of Edinburgh

21 September 2023
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 $n \mod 9$ (i.e. the remainder on dividing $n$ by 9).

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

$$
\begin{array}{cccccccccccc}
4 & 1 & 6 & 4 & 7 & 3 & 5 & 2 & 5 & 3 & 7 & 7 & 2 & 8 & 9 \\
9 & \overline{3} & 7 & 1 & 4 & 5 & 8 & 4 & 2 & 6 & 6 & 3 & 1 & 4 & 7 & 2 & 2 & 4 & 8 & 3 & 3 & 6 & 9 & 6 & 5 & 2 & 6 & 8 & 0 & 8 & 7 & \text{rem 6}
\end{array}
$$

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 $n = 3748261728395607$:

$$3+7+4+8+2+6+1+7+2+8+3+9+5+6+0+7 = 78$$
$$7 + 8 = 15$$
$$1 + 5 = 6$$

Why does this work?

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

$$1000a + 100b + 10c + d = (999a + a) + (99b + b) + (9c + c) + d$$
$$≡ a + b + c + d \text{ (mod 9)}$$
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.
Problem 2: Given (large) whole numbers $a, n, m$, compute $a^n \mod m$. E.g. $2^{10} \mod 17 = 1024 \mod 17 = 4$.

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:

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

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

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

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

IADS Lecture 2 Slide 6
Fast modular exponentiation

Method C: Notice that it’s easy to compute \( e = a^n \mod m \) if we’ve already computed \( d = a^\lfloor n/2 \rfloor \mod m \):

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

This suggests the following recursive algorithm:

\[
\text{Expmod} \ (a,n,m): \quad \# \text{ Computes } a^n \mod m
\]

if \( n=0 \) then return 1
else
    \( d = \text{Expmod} \ (a,\lfloor n/2 \rfloor,m) \)
    if \( n \) is even
        return \( (d \times d) \mod m \)
    else return \( (d \times d \times a) \mod m \)

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

\textbf{Expmod} \( (a,n,m) \):
if \( n=0 \) then return 1
else
\[ d = \text{Expmod} \ (a,\lfloor n/2 \rfloor,m) \]
if \( n \) is even
return \( (d \times d) \mod m \)
else return \( (d \times d \times a) \mod m \)

Imagine each evaluation of \textbf{Expmod} is done by a different ‘person’:

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

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

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

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

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 \( a^{n-1} \mod n = 1 \). [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 \( 2^{n-1} \mod n = \cdots \neq 1 \). 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).
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:

$I\text{NSort}(A)$:

\[
\begin{align*}
&\text{for } i = 1 \text{ to } |A| - 1 \quad \# |A| \text{ means size of } A \\
&\quad x = A[i] \\
&\quad j = i - 1 \\
&\quad \text{while } j \geq 0 \text{ and } A[j] > x \\
&\quad \quad j = j - 1 \\
&\quad A[j+1] = x
\end{align*}
\]
**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 $B[i] < C[j]$  

    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:

\[
\text{MergeSort} \ (A,m,n): \quad \# \text{ sorts } A[m], A[m+1], \ldots, A[n-1] \\
\quad \# \text{returning result in an array } D \text{ of size } n-m
\]

if \( n-m = 1 \)

\[
\text{return } [ \ A(m) \ ]
\]

else

\[
p = \lfloor (m+n)/2 \rfloor \\
B = \text{MergeSort} \ (A,m,p) \\
C = \text{MergeSort} \ (A,p,n) \\
D = \text{Merge} \ (B,C)
\]

return \( D \)

\[
\text{MergeSortAll} \ (A): \quad \text{return } \text{MergeSort} \ (A,0,|A|)
\]
Python again

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>InsertSort</th>
<th>MergeSort</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.023</td>
<td>0.068</td>
<td>0.34</td>
</tr>
<tr>
<td>100</td>
<td>0.97</td>
<td>0.74</td>
<td>1.31</td>
</tr>
<tr>
<td>1000</td>
<td>69.8</td>
<td>7.9</td>
<td>8.84</td>
</tr>
<tr>
<td>10000</td>
<td>8210.</td>
<td>76.2</td>
<td>107.</td>
</tr>
<tr>
<td>100000</td>
<td>906000.</td>
<td>1080.</td>
<td>839.</td>
</tr>
<tr>
<td>1000000</td>
<td>–</td>
<td>13300.</td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td>–</td>
<td>158000.</td>
<td></td>
</tr>
<tr>
<td>100000000</td>
<td>–</td>
<td>2619000.</td>
<td></td>
</tr>
</tbody>
</table>

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.]