Introduction to Algorithms and Data Structures

Tutorial 2

your tutor

School of Informatics
University of Edinburgh

7th-9th October, 2025

Q1: arithmetic operations of $\mathbf{Expmod}$

The task is to evaluate

$$a^n \bmod m \quad \text{as} \quad \mathbf{Expmod}(a, n, m) \text{ (different methods),}$$

and then from that

$$\mathbf{ProbablePrime}(n) \;=\; (\mathbf{Expmod}(2, n-1, n) \stackrel{?}{=} 1)$$

How many arithmetic operations $(\times, +, -, \mathrm{div}, \mathrm{mod})$ will be used?

Q1: arithmetic operations of $\mathbf{Expmod}$

The task is to evaluate

$$a^n \bmod m \quad \text{as} \quad \mathbf{Expmod}(a, n, m) \text{ (different methods),}$$

and then from that

$$\mathbf{ProbablePrime}(n) \;=\; (\mathbf{Expmod}(2, n-1, n) \stackrel{?}{=} 1)$$

How many arithmetic operations $(\times, +, -, \mathrm{div}, \mathrm{mod})$ will be used?

Method B:

```
Expmod (a,n,m):
    b=1
    for i = 1 to n
        b = (b × a) mod m
    return b
```

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
```

Q1 (a): Method B

Method B:

```
Expmod (a,n,m):
    b=1
    for i = 1 to n
        b = (b × a) mod m
    return b
```

- We do 3 arithmetic operations for each i ($\times$, mod, increment i)
- We do n iterations $\Rightarrow$ exactly $3n$ arithmetic operations. So $\Theta(n)$.

For **ProbablePrime**(n) (we have $a = 2$), we also have to subtract 1 from n to set up the computation $\mathbf{Expmod}(2, n-1, n)$ ... $\Rightarrow$
we have $3n + 1$ arithmetic operations, still $\Theta(n)$.

note 1: What about the other operations?
note 2: Constants for the $O(\cdot)$ and the $\Omega(\cdot)$?

Q1 (b): Method C

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
```

Q1 (b): Method C

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
```

- 3-4 arithmetic operations at each "level" ($\times$, mod, div, maybe a 2nd $\times$)
- We visit *at most* $\lg n$ (ie $\log_2 n$) recursive "level" $\Rightarrow$ $\leq 4 \lg n$ arithmetic operations. So $O(\lg n)$.

For the $\Omega(n)$ counterpart consider the case when n is a power of 2.
For **ProbablePrime**(n) again $\Theta(n)$ transfers directly (just 1 extra operation).

note: Constants?

Q2: Bubblesort

```
BubbleSort (A):
    for i = 1 to |A| − 1
        for j = 0 to |A|−2
            if A[j] > A[j+1]
                swap A[j] and A[j+1]
```

- Run an example with (say) $18, 11, 2, 9, 8$
- Discuss the “sweeps”

Bubblesort example

```
BubbleSort (A):
    for i = 1 to |A| − 1
        for j = 0 to |A|−2
            if A[j] > A[j+1]
                swap A[j] and A[j+1]
```

Q2(a) - correctness

claim: *After the first sweep through the array, the largest element will be in its correct place at position $|A| - 1$ (ie, $n - 1$).*

- Suppose the largest element starts at position k.
  - If $k = |\mathrm{A}| - 1$, then A[k] is already in the correct place.
    It will not be moved (explain why).
  - If $k < |\mathrm{A}| - 1$, then when $j = k$, this larger element will be swapped into position $k + 1$.
    For *every* $j = k + 2, \ldots$, this largest value gets swapped right until j reaches $|\mathrm{A}| - 2$, when the element gets swapped into $|\mathrm{A}| - 1$.

claim: *After $|A| - 1$ sweeps (ie $n - 1$ sweeps), the array will be fully sorted.*

"Invariant" is that after i sweeps, the largest i elements $x_1 \leq x_2 \leq \cdots \leq x_i$ are in sorted order in the top i positions.

note: Talk about invariants.

Q2 (b) - asymptotic running-time

Asymptotic worst- and best-case number of comparisons for **BubbleSort**.

For **BubbleSort**, we *always* do a sequence of $(n-1)$ "sweeps" up the arrays:

- Each "sweep" is of length n, with $n-1$ comparisons, maybe some swaps.
- As "sweeps" progress, the big items gather (in sorted order) up the top.
- In the later sweeps, most comparisons are redundant, few swaps happen.

The worst running-time and best-case running-time are *both* in proportion to $(n-1)^2$: on *all* inputs of length n, the algorithm performs exactly $(n-1)^2$ comparisons (even if no swaps take place).

Best-case is already-sorted order, worst-case reverse-sorted order.

Only affects swaps though. And overall both cases are $\Theta(n^2)$

Q2(c) - improvements

Improvement 1: After sweep i, items in positions |A|−i, ..., |A|−1 never move.
⇒ so upper limit of j should be $|A| - i - 1$, not $|A| - 2$

Improvement 2: If we do a sweep with 0 swaps, the array is sorted.
⇒ count swaps each sweep, terminate when it hits 0 ('flg' boolean variable).

```
BubbleSort2(A):
    i = 1
    repeat
        i = i+1
        flg = false
        for j = 0 to |A|−i
            if A[j] > A[j+1]
                swap A[j] and A[j+1]
                flg = true
    until flg = false
```

Q2 (d) - asymptotics of Bubblesort2

The **worst-case** number of comparisons has roughly halved (now $n(n-1)/2$), but is still $\Theta(n^2)$.

worst-case occurs when the input A is reverse-sorted.

- We will always do $|A| - 1 - i$ swaps on the i-th iteration (the highest item for 0, 1, . . ., $|A| - 1 - i$ is in position 0).
- So we run every "sweep" (length n, $n - 1$, . . ., 2)
- $\sum_{i=1}^{n-1} i = \Theta(n^2)$ like Insertsort.

best case occurs when A is already sorted.

- Now the *first* sweep does 0 swaps, so we can stop immediately.
- $\Theta(n)$ best-case running-time.

Q2(e) - challenge

claim: *The number of comparisons performed by* **BubbleSort2** *on input A is at least the "unsortedness" of A.*

(the "unsortedness" is the count of i,j ($i < j$) pairs such that A[i] $>$ A[j])

Proof: Let i,j be an inversion in the input A, i.e. initially A[i] $=$ x $>$ y $=$ A[j].

Track the movements of x and y as the computation proceeds.

- At the start we have x before y ...
- and at the end (when A is sorted) we must have x after y.
- But x (and y) can move by only one position each step, so ...
  there must be a time when these elements meet and are swapped;
  (and for this to happen, they must have been directly compared)

So for any "inversion" i,j we will see a specific comparison of these items (and it can only be this exact inversion). Therefore

$$\text{number of comparisons} \;\geq\; \text{number of inversions.}$$

Q3: Write a version of MergeSort that uses just two arrays A and B of size n.

John's solution: We require two subroutines:

- **MergeAtoB**(m,p,n): merges the segment A[m],...,A[p−1] with the segment A[p],...,A[n−1]
  (assuming these segments are themselves already sorted),
  and writes the result to B[m],...,B[n−1].
- **MergeBtoA**(m,p,n): merges the segment B[m],...,B[p−1] with the segment B[p],...,B[n−1], and writes the result to A[m],...,A[n−1].

Minor variants of the **Merge** procedure from lectures (except do not return a value). Should work correctly even when one of the segments has length 0.

Q3: John's solution cont'd.

The following recursive procedure for MergeSort will then work:

MergeSort(m,n):
  if $n - m > 1$
    $q = \lfloor (m+n)/2 \rfloor$
    $p = \lfloor (m+q)/2 \rfloor$
    $r = \lfloor (q+n)/2 \rfloor$
    MergeSort(m,p)
    MergeSort(p,q)
    MergeSort(q,r)
    MergeSort(r,n)
    MergeAtoB(m,p,q)
    MergeAtoB(q,r,n)
    MergeBtoA(m,q,n)

Q3 cont'd: What is the memory space use of this algorithm?

The arrays A and B (together) occupy $\Theta(n)$ of memory: main space requirement.

We also need to keep track of certain information for each of the recursive calls to **MergeSort** currently in progress:

- values of m,n,q,p,r,
- plus a record of which line of code we've got to in that call, so that we know where to return to.

This is $\Theta(1)$ of information per call.

The maximum depth of recursion is $\lceil \log_4(n) \rceil$, so $\Theta(\lg n)$ of memory altogether. (In a typical programming language implementation, all this information will be stored on the *call stack*.)

While a call to **MergeAtoB** or **MergeBtoA** is in progress, there will also be the variables i,j,k associated with this call: just $\Theta(1)$ space.

So the total memory requirement is $\Theta(n) + \Theta(\lg n) + \Theta(1) = \Theta(n)$.

Q3: Alternative solution.

note: Main improvement in 4-way solution is coming from the switch from Merge/MergeAtoB becoming procedures (where sorted output resides in A) instead of functions (where output gets saved into new sub-array).

- We can take the same approach for a 2-way split.
- A is the input array (also where the sorted output is saved) of size n, B is the "scratch array" of same size.
- We set up Merge as **MergeAtoB**. We also have an extra method called **copyBtoA**.

Q3: Alternative solution.

Two global arrays A (input/output array), and B (scratch array)

MergeSort(m,n):
    if n$-$m $>$ 1
        p $= \lfloor$(m+n)/2$\rfloor$
        MergeSort(m,p)
        MergeSort(p,n)
        MergeAtoB(m,p,n)
        copyBtoA(m,n)

Q3 cont'd: What is the memory space use of this algorithm?

The arrays A and B take $\Theta(n)$ of memory: like John's algorithm.

Information for each of the recursive calls to **MergeSort** in progress:

- values of m,p,n,
- plus a record of which line of code we've got to in that call, so that we know where to return to.

This is $\Theta(1)$ of information per call.

The maximum depth of recursion is $\lceil \lg(n) \rceil$, so $\Theta(\lg n)$ of memory altogether. Similar/fewer variables i,j,k for **mergeAtoB**/**copyBtoA** calls to store on stack.. So the total memory requirement is $\Theta(n) + \Theta(\lg n) + \Theta(1) = \Theta(n)$.

John comment: 4-way method avoids extra copyings that don't do any merging: if we expand the 2-way method 2 levels, it does the same merges as John's but also some copyings. Increases the leading coefficient inside the O just a bit.