Analysis of Algorithms... How to?

An algorithm should work for (input) data of any size.
(Example TableLookUp: input size is the size of the table.)

Show the resource (time/memory) used as an increasing function of input size.

Focus on the worst case performance!

Ignore constant factors

- analysis should be machine-independent;
- more powerful cpu \(\rightarrow\) speed-up by constant factors.

Study scalability / asymptotic behaviour for large problem sizes: ignore lower-order terms, focus on dominating terms.

Estimating execution time for iterative programs

- Elementary operation
takes/can be bound by a constant time

- Sequence of operations
takes the sum of the times of its components

- Loop (for... and while...)
the time of the body multiplied by number of repetitions (in the worst case)
or: \( t_{\text{loop}} = \sum_{i=1}^{n} t_i \) for maximum \( n \) where \( t_i \) are the times of the body

  Remark: loops are a generalization of sequences

  Remark: to know \( n \) is to know that the loop terminates

- Conditional statement (if...then...else...)
the time for evaluating and checking the condition
plus maximum of the times for then and else parts.
Different techniques:

Issue: growth rate of...
- Memory (stack, allocated data structures)
- Execution time

Situation to analyze:
- worst case, best case, expected case, amortized: a sequence of calls of the algorithm

Techniques:
- Algebraically, (count iterations, see last slide)
- Recursive algorithms (solve recurrence relations)
- Probabilistic analysis (figure out average case behaviour)

Example 2 of "algebraic" analysis...

```plaintext
function BinarySearch(table T [1:n], key K) : Integer
(0) if n ≤ 0 then return −1
(1) i, u ← 0, n − 1
(2) while i < u do
(3)   mid ← (i + u)/2
(4)   if K = T[mid] then return mid
(5)   if K < T[mid] then u ← mid − 1 else i ← mid + 1
(6) if K = T[i] then return i else return −1

Worst case time: \( t_s + t_f + \text{maxit} \cdot (t_1 + t_3 + t_4) + t_b \)
where maxit = maximal number of iterations of the while loop
```

Example: Independent Nested loops...

Matrix-vector product (here, for a quadratic matrix)

- given: vector \( \mathbf{x} \in \mathbb{R}^n \), matrix \( \mathbf{A} \in \mathbb{R}^{n \times n} \) with \( n > 0 \)
- compute: vector \( \mathbf{y} \in \mathbb{R}^n \) with
  \[
  \mathbf{y} = \mathbf{A} \mathbf{x}
  \]
  \[
  \text{That is, } y_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = 1,...,n
  \]

```plaintext
procedure matrix-vector \( \mathbf{A} \mathbf{x} \rightarrow \mathbf{y} \)\n
(1) for \( i \) from 1 to \( n \) do
(2) \( y_i \leftarrow 0.0 \)
(3) for \( j \) from 1 to \( n \) do
(4) \( y_i \leftarrow y_i + a_{ij} x_j \)

return \( \mathbf{y} \)
```

Example: Dependent Nested Loops

Prefix-Sums

- given: Vector \( \mathbf{x} \in \mathbb{N}^n \)
- compute: "Prefix-sums" vector \( \mathbf{y} \in \mathbb{N}^n \) with
  \[
  y_i = \sum_{j=1}^{i} x_j, \quad i = 1,...,n
  \]

A straightforward algorithm follows directly from the definition:

```plaintext
procedure prefix-sum \( \mathbf{x} \rightarrow \mathbf{y} \)

(1) for \( i \) from 1 to \( n \) do
(2) \( y_i \leftarrow 0.0 \)
(3) for \( j \) from 1 to \( i \) do
(4) \( y_i \leftarrow y_i + x_j \)

return \( \mathbf{y} \)
```
Example: Dependent Nested Loops (cont.)

- Att ta på tavlan: 
  ...även vid kränliga indexgränsar blir det något liknande...

\[
T(n) = \sum_{i=1}^{n} (t_i + t_2 + \sum_{j=1}^{i-1} (t_j + t_3)) \\
= n(t_1 + t_2) + \frac{n(n+1)}{2} (t_i + t_3) \\
= n(t_1 + t_2) + \frac{n^2 + n}{2} (t_i + t_3) \\
= n(t_1 + t_2) + \frac{n^2 (t_1 + t_3) + n(t_3 + t_4)}{2} \\
= n \left( \frac{2i + 2i_2 + i_3 + t_3}{2} \right) + i_3 \\
= n \left( \frac{2i + 2i_2 + i_3 + t_3}{2} \right) + i_3 \\
= \frac{2i + 2i_2 + i_3 + t_3}{2} + i_3 \\
= \frac{2i + 2i_2 + i_3 + t_3}{2} + i_3 \\
= \frac{2i + 2i_2 + i_3 + t_3}{2} + i_3 \\
= \frac{2i + 2i_2 + i_3 + t_3}{2} + i_3 \\
= \frac{2i + 2i_2 + i_3 + t_3}{2} + i_3 \\
= \sum_{i=1}^{n} \left( \frac{2i + 2i_2 + i_3 + t_3}{2} + i_3 \right)
\]

Analysis of Recursive Programs...

- Characterize execution time by a recurrence relation
- Find solution (closed form, non-recursive) of the recurrence relation

If not listed in a textbook, you may:
1. Unroll the recurrence relation a few times to get a hypothesis for a possible solution: \( T(n) = \ldots \)
2. Prove the hypothesis for \( T(n) \) by mathematical induction. If that fails, modify the hypothesis and try again...

Analysis of Recursive Program (1)

```plaintext
function fact(n) : integer
if n = 0 then return 1
else return n * fact(n-1)
end function
```

Execution time:
- Time for comparison: \( t_c \)
- Time for multiplication: \( t_m \)
- Time for call and return: \( t_r \)

Total execution time \( T(n) \) is defined by a recurrence relation:

\[ T(0) = t_c \]
\[ T(n) = t_c + t_m + T(n-1), \text{ if } n > 0 \]

Hence for \( n > 0 \):

\[ T(n) = (t_c + t_m) + (t_c + t_m) + T(n-2) \]

\[ = (t_c + t_m) + (t_c + t_m) + T(n-3) \]
\[ = (t_c + t_m) + (t_c + t_m) + \ldots + t_c \]
\[ = n(t_c + t_m) + c \in O(n) \]

Towers of Hanoi

Task: Move pile of bricks to an empty pole

Limitations:
- Smaller bricks on top
- Move one brick at a time
Towers of Hanoi...

procedure Hanoi(integer n, char X,Y,Z):  
{ move n topmost slices from tower X to tower Z, using Y as temporary }  
if n = 1 then output(“move X to Z”)  
else  
   Hanoi(n-1, X, Z, Y)  
   output(“move X to Z”)  
   Hanoi(n-1, Y, Z, X)  
return

As stated earlier:  
• Formulate an equation T(1)=... T(2)=...  
• Unroll a few times, get hypothesis for T(n)=...  
• Prove the hypothesis!

Towers of Hanoi — run on black board!

The experiment...  

\begin{align*}  
T(1) &= c \\
T(2) &= T(1) + c + T(1) = 3c \\
T(3) &= T(2) + c + T(2) = 7c \\
T(4) &= T(3) + c + T(3) = 15c \\
T(5) &= T(4) + c + T(4) = 31c \\
\vdots \end{align*}

The hypothesis:

\[ T(n) = 2^n - 1 \]

The proof:

\begin{align*}  
T(n+1) &= T(n) + c + T(n) \\
&= 2(2^n - 1) + c + 2^{n-1}c - 2c + c \\
&= 2^{n+1}c - c = 2^{n+1}c - 1c \\
&= 2^{n+1}c - 1c \\
&= 2^{n+1}c - 1c
\end{align*}

Therefore: Hanoi \( \in \mathcal{O}(2^n) \)

Average analysis

Reconsider TableSearch(): sequential search through a table  
• Input argument: one of the table elements,  
• Assume it is chosen with equal probability for all elements.

function TableSearch(table<key> T, key K): Integer  
for i from 0 to n-1 do  
if T[i] = K then return i

Expected search time:

\[ t_1 + 2t_2 + 3t_3 + \ldots + nt_n = \frac{(1+2+3+\ldots+n)n}{n} = \frac{n(n+1)}{2n} \times t_1 = \frac{n+1}{2} \in \mathcal{O}(n) \]

Amortized Analysis

• Example: dynamic array implementation (e.g., flexible string buffer representation)

• Init(): a 1 element empty array is created

• Append(c): Character c is appended at the end  
  \hfill \text{if buffer is full:}  
  \begin{itemize}  
  \item allocate a new 2 times as big  
  \item Copy data to the new one  
  \end{itemize}

\begin{tikzpicture}[baseline=-0.5ex]  
\node[draw] (array) {CurrSize: 8 Limit: 8};  
\end{tikzpicture}
Amortized Analysis (cont.)

What is the time complexity of `Append(c)`?

- Most cases, $O(1)$
- When we re-allocate
  - Copying of $n$ data elements $O(n)$
  - Insertion of the new element $O(1)$

Amortized Analysis – charge ahead

Goodrich/Tamassia page 229-230

- Dynamic Vector – reallocate on overflow:
  - Reallocate a new vector with double size (cost: 0#)
  - Copy all items from old vector
  - Insert the item that caused the overflow

- Actual "cost" to insert/copy a value: 1#

- Charge user 3# for each call to `insertLast`

- Start with empty vector:

Amortized Analysis – example

- `insertLast("A")`
  - overflow, allocate 1, insert 1
- `insertLast("B")`
  - overflow, allocate 2, copy 1, insert 1
- `insertLast("C")`
  - overflow, allocate 4, copy 2, insert 1
- `insertLast("D")`
  - no overflow, insert 1
- `insertLast("E")`
  - overflow, allocate 8, copy 4, insert 1
- `insertLast("FGH")`
  - no overflow, insert 1 x 3