Estimated Time to Sum Subarrays

- Ignore data initialisation
- "Brute-force" summing with two nested loops:

$$T(n) = m(m+1) = {}^{n}/_{2}({}^{n}/_{2} + 1)$$
$$= 0.25n^2 + 0.5n$$

- For a large n, $\mathrm{T}(n) \cong 0.25n^2$
  - e.g., if $n \geq 10$, the linear term $0.5n \leq 16.7\%$ of $\mathrm{T}(n)$
  - if $n \geq 500$, the linear term $0.5n \leq 0.4\%$ of $\mathrm{T}(n)$

Quadratic vs linear term

$T(n) = 0.25n^2 + 0.5n$				
n	$T(n)$	$0.25n^2$	$0.5n$	
10	30	25	5	16.7%
50	650	625	25	3.8%
100	2550	2500	50	2.0%
500	62750	62500	250	0.4%
1000	250500	250000	500	0.2%

Quadratic Time to Sum Subarrays: $T(n)=0.25n^2 + 0.5n$

- Factor $c = 0.25$ is referred to as a "**constant of proportionality**"
- An actual value of the factor does not effect the behaviour of the algorithm for a large n:
  - 10% increase in $n \rightarrow$ 20% increase in $T(n)$
  - Double value of $n \rightarrow$ 4-fold increase in $T(n)$:

$$T(2n) = 4\ T(n)$$

Running Time: Estimation Rules

- Running time is proportional to the **most significant term** in $T(n)$
- Once a problem size becomes large, the most significant term is that which has the largest power of n
- This term increases faster than other terms which reduce in significance

Running Time: Estimation Rules

- Constants of proportionality depend on the compiler, language, computer, etc.
  - It is useful to ignore the constants when analysing algorithms.
- Constants of proportionality are reduced by using faster hardware or minimising time spent on the "inner loop"
  - *But this would not effect behaviour of an algorithm for a large problem!*

Elementary Operations

- Basic arithmetic operations (**+** ; – ; **∗** ; / ; **%**)
- Basic relational operators (==, **!**=, >, <, >=, <=)
- Basic Boolean operations (AND,OR,NOT)
- Branch operations, return, …

Input for problem domains (meaning of n):

<u>Sorting</u>: n items

<u>Graph / path</u>: n vertices / edges

<u>Image processing</u>: n pixels

<u>Text processing</u>: string length

Estimating Running Time

- **Simplifying assumptions:**
  all elementary statements / expressions take the same amount of time to execute
  - e.g., simple arithmetic assignments
  - `return`
- Loops increase in time **linearly** as

$$k \cdot T_{\text{body of a loop}}$$

where k is number of times the loop is executed

Estimating Running Time

- Conditional / switch statements like **if** {*condition*} **then** {*const time* T_1} **else** {*const time* T_2} are more complicated (one has to account for branching frequencies: $T = f_{\mathrm{true}} T_1 + (1 - f_{\mathrm{true}}) T_2 \leq \max\{T_1, T_2\}$
- **Function calls**:

$$T_{\text{function}} = \Sigma\, T_{\text{statements in function}}$$

- **Function composition**:

$$T(f(g(n))) = T(g(n)) + T(f(n))$$

Estimating Running Time

- Function calls in more detail: $T = \sum T_{\text{statement } i}$

… x.myMethod(5, …);

public void myMethod(int a, …){

statements $1, 2, \dots, N$ }

- Function composition in more detail: $T(f(g(n)))$

Computation of $x = g(n) \rightarrow T(g(n))$

Computation of $y = f(x) \rightarrow T(f(n))$

Example 1.6: Textbook, p.13

Logarithmic time due to an exponential change $i = k, k^2, k^3, \ldots, k^m$ of the loop control in the range $1 \le i \le n$:

for $i = k$ **step** $i \leftarrow ik$ **until** n **do**
… {const # of elementary operations}
end for

m iterations such that $k^{m-1} < n \le k^m \Rightarrow$
$T(n) = c \lceil \log_k n \rceil$

Example 1.7: Textbook, p.13

$n \log n$ running time of the conditional nested loops:

m $\leftarrow 2$; **for** $j \leftarrow 1$ **to** n **do**

if $(j = m)$ **then**

$m \leftarrow 2m$

for $i \leftarrow 1$ **to** n **do** …{const # of operations}

end for

end if

end for

The inner loop is executed k times for $j = 2, 4, \ldots, 2^k$;

$k < \log_2 n \leq k + 1$; in total: $\boxed{T(n) = kn = n \lfloor \log_k n \rfloor}$

Exercise 1.2.1: Textbook, p.14

Conditional nested loops: linear or quadratic running time?

$m \leftarrow 1;$ **for** $j \leftarrow 1$ **to** n **do**

if $(j = m)$ **then** $m \leftarrow m\,(n - 1)$

for $i \leftarrow 1$ **to** n **do** …{const # of operations}

end for

end if

end for

The inner loop is executed only twice, for $j = 1$ and $j = n - 1$;

in total: $T(n)=2n$ → **linear** running time