Slide 1

Computing 1A (COMP1011)

Work complexity #2

Manuel M. T. Chakravarty

Slide 2

Work Complexity—Revision

The motivation:

- ➜ Different solutions to a programming problem can have different work complexity (resulting in different performance)
- ➜ Compare programs by inspecting the source (without benchmarking)

The method:

- ➜ Count reduction steps: primitive operations & function calls
- ➜ Derive a timing function $T(n)$
- ➜ It gives the number of required reductions steps in dependence on the size n of the input
- ➜ We distinguish
  - worst case: input of a form (not size!) that needs minimal work
  - best case: input of a form that needs maximal work
  - average case: randomised input

Slide 3

For recursive functions, we proceed in three steps:

① Calculate the steps in dependence of a variable input size n

② Simplify the resulting formula

③ Bring the resulting recurrence relation into a closed form

Example: worst case complexity of `isort`:

```
isort          :: Ord a => [a] -> [a]
isort []        = []
isort (x:xs)    = insertSorted x (isort xs)
```

① $T^{worst}_{isort}(0) = 1$

② $T^{worst}_{isort}(n) = 1 + T^{worst}_{insertSorted}(n-1) + T^{worst}_{isort}(n-1)$

$= 1 + 2(n-1) + T^{worst}_{isort}(n-1)$

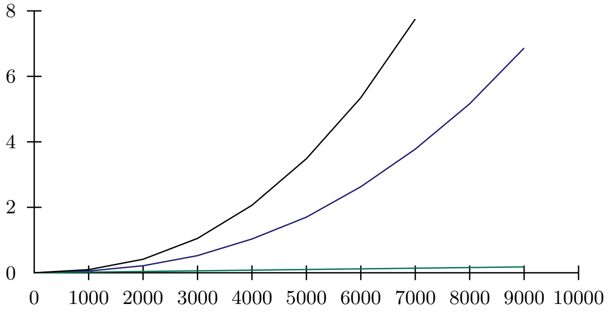
③ $= 2n - 1 + T^{worst}_{isort}(n-1)$

$= (\sum_{i=1}^{n}(2i-1)) + 1 = 2 * \sum_{i=1}^{n} i - (\sum_{i=1}^{n} 1) + 1$

$= 2(n(n+1)/2) - n + 1 = n^2 + 1$

Slide 4

Running Times of `isort` on a randomly generated list, on a sorted list and a list in reverse order:



Asymptotic Complexity ($\mathcal{O}$-Notation)

Slide 5

Summary of `isort`:

$$T^{best}_{isort}(n) = 3 * n + 1$$
$$T^{worst}_{isort}(n) = n^2 + 1$$
$$T^{av}_{isort}(n) = 0.5 * n^2 + 0.5 * n$$

Observations:

- Formulas match curves determined by benchmarking
- But is this level of detail really needed?
- How careful must basic steps be defined and exact number of reduction steps be counted?
- After all we already abstract over machine details, such as caches etc.

Focus on the growth rate of the running time as function of input size, rather than get bogged down with small details.

Slide 6

Assumptions concerning complexity functions:

- the arguments are restricted to non-negative numbers
- the function values are non-negative for all arguments

Intuition: Instead of saying

- `isort` requires $0.5n^2 + 0.5n$ steps on average for lists of length n

we say

- `isort` has an asymptotic work complexity of $\mathcal{O}(n^2)$ for a list of length n

or, most of the times just

- `isort` is $\mathcal{O}(n^2)$

quietly assuming that n is the no of elements to be sorted and that we are only considering the average case.

Slide 7

The $\mathcal{O}$-notation categorises complexity functions into classes:

Definition: Let $T(n)$ be a timing function, then $T(n)$ is $\mathcal{O}(f(n))$ if there exisits

- a non-negative number n_0, and
- a non-negative number c, such that for all $i > n_0$

$$T(i) \leq c * f(i)$$

Slide 8

Is $T^{best}_{isort}(n) = 3 * n + 1$ in $\mathcal{O}(n)$?

- Same as: Can we find n_0 and c such that for all $n > n_0$

$$3 * n + 1 \leq c * n$$

- Yes! For $c = 4$ and $n_0 = 2$:

$$3 * n + 1 \leq 4 * n \quad \text{is true for all } n > 2$$

- $c = 4$ and $n_0 = 2$ are called witnesses to the fact that $T^{best}_{isort}(n)$ is $\mathcal{O}(n)$

Slide 9

Is $T_{isort}^{av}(n) = 0.5n^2 + 0.5n$ in $\mathcal{O}(n)$?

- ➜ Same as: Can we find n_0 and c such that for all $n > n_0$

$$0.5n^2 + 0.5n \leq c * n$$

- ➜ No! Because $T_{isort}^{av}(n)$ will eventually "overtake" $c * n$, no matter how big c is.
- ➜ But it is possible to show that $T_{isort}^{av}(n)$ is $\mathcal{O}(n^2)$

Slide 10

Is $T_{isort}^{best}(n) = 3 * n + 1$ in $\mathcal{O}(n^2)$?

- ➜ Same as: Can we find n_0 and c such that for all $n > n_0$

$$3 * n + 1 \leq c * n^2$$

- ➜ Yes! Previous witnesses $c = 4$ and $n_0 = 2$ work here as well:

$$3 * n + 1 \leq 4 * n^2 \quad \text{is true for all } n > 2$$

- ➜ Every function that is $\mathcal{O}(n)$ is also $\mathcal{O}(n^2)$
- ➜ $\mathcal{O}$-notation gives us only an upperbound!

Slide 11

Summary:

To prove that a program has asymptotic complexity $\mathcal{O}(f)$, we

1. determine the parameter(s) the program's work complexity depends on,
2. derive the timing function $T(n)$, and
3. show that $T(n)$ is in $\mathcal{O}(f)$ by providing witnesses n_0 and c.

Slide 12

Some Important Classes:

$\mathcal{O}$	Verbal form
$\mathcal{O}(1)$	Constant
$\mathcal{O}(\log n)$	Logarithmic
$\mathcal{O}(n)$	Linear
$\mathcal{O}(n \log n)$	Quasi-linear
$\mathcal{O}(n^2)$	Quadratic
$\mathcal{O}(n^3)$	Cubic
$\mathcal{O}(2^n)$	Exponential

Slide 13

To classify a function, we are usually looking for the "tightest" and "simplest" $\mathcal{O}$ upperbound we can prove.

Rule of Thumb:

- ➜ Drop all constant factors
  - $\mathcal{O}(1)$ instead of $\mathcal{O}(7)$
  - $\mathcal{O}(n)$ instead of $\mathcal{O}(7 * n)$
- ➜ Drop all functions in a sum that are dominated by other functions
  - In a polynomial, throw away all the terms but the one with the highest exponent:
  - $\mathcal{O}(n^4)$ instead of $\mathcal{O}(n^4 + n^3 + 1)$
  - $\mathcal{O}(2^n)$ instead of $\mathcal{O}(2^n + n^{10})$

Slide 14

Examples:

```
take:: Int -> [a] -> [a]
take 0 xs     = []
take n []     = []
take n (x:xs) = x : (take (n - 1) xs)
```

- ➜ `take` depends on two parameters!
- ➜ m: length of the list
- ➜ n: number of elements retrieved from the list

$$
\begin{aligned}
T_{\texttt{take}}(0,m) &= 1\\
T_{\texttt{take}}(n,0) &= 1\\
T_{\texttt{take}}(n,m) &= 3 + T_{\texttt{take}}(n-1,m-1)\\
&= 3 * min(m,n) + 1
\end{aligned}
$$

- ➜ $T_{\texttt{take}}(n,m) \leq 3 * n + 1$ ➡ $T_{\texttt{take}}(n,m)$ is in $\mathcal{O}(n)$
- ➜ $T_{\texttt{take}}(n,m) \leq 3 * m + 1$ ➡ $T_{\texttt{take}}(n,m)$ is in $\mathcal{O}(m)$
- ➜ `take` is $\mathcal{O}(n)$, $\mathcal{O}(min(m,n))$ and $\mathcal{O}(m)$
- ➜ since typically $n < m$, we often assume `take` is $\mathcal{O}(n)$

Slide 15

Another example:

```
substring:: String -> String -> Bool
substring ""    str  = True
substring str   ""   = False
substring str1  str2 =
  (prefix str1 str2) || (substring str1 (tail str2))

prefix:: String -> String -> Bool
prefix "" str = True
prefix str "" = False
prefix (s1:str1) (s2:str2)
  | s1 == s2  = prefix str1 str2
  | otherwise = False
```

Slide 16

Let n be length of first string, m length of second. Timing functions for the worst case:

- ➜ $T_{\texttt{pre}}(n,m)$
- ➜ $T_{\texttt{sub}}(n,m)$

```
prefix "" str = True
prefix str "" = False
prefix (s1:str1) (s2:str2)
  | s1 == s2  = prefix str1 str2
  | otherwise = False
```

$$
\begin{aligned}
T_{\texttt{pre}}(0,m) &= 1\\
T_{\texttt{pre}}(n,0) &= 1\\
T_{\texttt{pre}}(n,m) &= 2 + T_{\texttt{pre}}(n-1,m-1) = 2 * min(n,m) + 1
\end{aligned}
$$

```
substring ""    str  = True
substring str   ""   = False
substring str1  str2 =
   (prefix str1 str2) || (substring str1 (tail str2))
```

$$T_{\texttt{sub}}(0,m) = 1$$
$$T_{\texttt{sub}}(n,0) = 1$$

Slide 17

$$\begin{aligned} T_{\texttt{sub}}(n,m) &= 1 + T_{\texttt{pre}}(n,m) + T_{\texttt{sub}}(n,m-1) \\ &= (1 + 2 * min(n,m) + 1) * m + 1 \\ &= 2 * min(n,m) * m + 2 * m + 1 \end{aligned}$$

Let n be length of first string, m length of second. Timing function for the worst case:

$$T_{\texttt{pre}}(0,m) = 1$$
$$T_{\texttt{pre}}(n,0) = 1$$
$$T_{\texttt{pre}}(n,m) = 2 + T_{\texttt{pre}}(n-1,m-1) = 2 * min(n,m) + 1$$
$$T_{\texttt{sub}}(0,m) = 1$$
$$T_{\texttt{sub}}(n,0) = 1$$

Slide 18

$$\begin{aligned} T_{\texttt{sub}}(n,m) &= 1 + T_{\texttt{pre}}(n,m) + T_{\texttt{sub}}(n,m-1) \\ &= (1 + 2 * min(n,m) + 1) * m + 1 \\ &= 2 * min(n,m) * m + 2 * m + 1 \end{aligned}$$

- ➜ $T_{\texttt{sub}}(n,m)$ is $\mathcal{O}(min(n,m) * m)$
- ➜ $T_{\texttt{sub}}(n,m)$ is $\mathcal{O}(m^2)$
- ➜ $T_{\texttt{sub}}(n,m)$ is $\mathcal{O}(n * m)$