Slide 1

Computing 1A (COMP1011)

Work complexity #1

Manuel M. T. Chakravarty

Slide 2

Today's Lecture - Work Complexity of Programs:

- ➜ Analysis of Programs
- ➜ Running Time of Programs
- ➜ $\mathcal{O}$- Notation

Slide 3

Work Complexity of Programs

- ➜ Many programming problems have more than one correct solution
- ➜ For example, numerous algorithms to sort elements of a list
- ➜ How can we evaluate which of the possible solutions is the best?

Criteria we discussed so far:

- ➜ Design
- ➜ Clarity
- ➜ Structure

Another important criterion is the performance of a program.

Slide 4

How can we assess program performance?

- ➜ By measuring factors such as its
  - running time and
  - memory consumption
- ➜ In languages with automatic memory management, the two tend to be correlated
- ➜ More abstractly, we often speak of work complexity

Let's determine the running time for the sorting algorithm `isort` discussed before.

Slide 5

Example - Sorting Algorithm isort

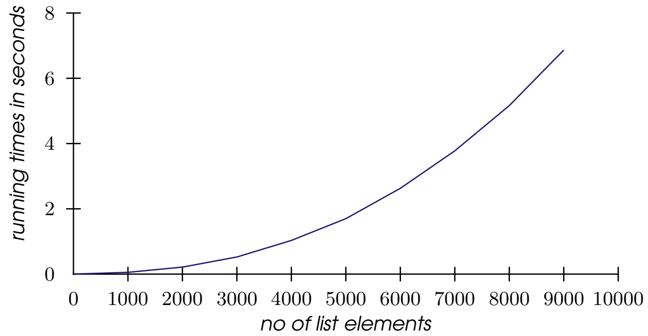
```
isort         :: Ord a => [a] -> [a]
isort []       = []
isort (x:xs)   = insertSorted x (isort xs)

insertSorted          :: Ord a => a -> [a] -> [a]
insertSorted e []     =  [e]
insertSorted e (x:xs)
  | e < x       = e : x : xs
  | otherwise   = x : insertSorted e xs
```

- What are the running times for different size input lists?

Slide 6

Running Times of `isort` on a randomly generated list:



- Doubling the list size squares running times:
  - 2000 elements: 0.21 s, 4000 elements: 1.03 s
  - 3000 elements: 0.52 s, 6000 elements: 2.62 s
- Running times grow quadratically!

Slide 7

How can we decrease the running times for `isort`?

- Use a faster processor?
- Compiler instead of interpreter?
- Other language?

Problems with these approaches:

- Make the program faster, but only by a constant factor
- For example, a processor with twice the speed does not even allow for an input list of twice the length
- It's the `isort` algorithm itself that implies quadratically increasing running times
- How can we determine the relative efficiency of programs?

- It seems as if we need to compare running times.

Slide 8

Analysing Programs

Can we estimate the running time of a program by just looking at the code?

- Not possible to predict the absolute running time in seconds
- Depends on too many factors:
  - processor speed,
  - caches,
  - memory,
  - compiler, etc.
- But we should able to see if the running time grows linearily, quadratically etc.

What exactly can we determine by just looking at the code?

Slide 9

Symbolic analysis of program code:

- ➜ Determine a function $T(n)$ that estimates the running time
- ➜ $T(n)$ depends on the size of the input n
- ➜ We count the number of reduction steps or primitive operations performed

Properties:

- ➜ Useful to evaluate the relative efficiency of any two algorithms
- ➜ Independent from the hardware and software environment
- ➜ Even studying a high-level description of an algorithm without actually implementing or running it
- ➜ Basic steps can vary slightly depending on interpretation

➨ We need to define what exactly our basic steps or primitive operations comprise before we can start analysis.

Slide 10

Set of Basic Steps:

① every arithmetic or boolean operation is defined as exactly one basic step e.g.
  - `5 + 7`
  - `x <= 10`

② every constructor application takes one step
  - `x:xs`
  - `Circle r v`

③ application of a function requires one basic step plus number of steps the evaluation of the function itself takes
  - `tail xs`

Slide 11

Example: How many steps does the evaluation of `insertSorted 5 [1, 3, 6]` take?

Required reduction steps:

```
  insertSorted 5 [1, 3, 6]
=  {insertSorted.3 (stepping case)}
  1 : (insertSorted 5 [1, 3, 6])
=  {insertSorted.3 (stepping case)}
  1 : (2 : (insertSorted 5 [6]))
=  {insertSorted.2 (base case 2)}
  1 : 2 : 5 : [6]
```

Required comparisons:

- ➜ We evaluate `5 > 1`, `5 > 2`, and `5 > 6`
- ➜ Three basic operations

➨ Evaluation of the expression takes 6 basic steps

Slide 12

How many steps does `isort` require?

- ➜ We want to know the number of steps required
- ➜ Not only for a fixed input
- ➜ Relation between input size and no. of steps
- ➜ We determine the function $T(n)$ which gives us the number of steps for a given list size n.

Since `isort` uses `insertSorted`, we start by determining a timing function $T(n)$ for `insertSorted`

Slide 13

Running time of `insertSorted`

- assume we insert an element into a list of length n
- $n = 0$: only a single step, i.e.,

  $$T(0) = 1$$

- otherwise, it depends on the values in the list!
  - best case: new element smaller than first element in the list
  - worst case: new element greater than last element in the list
  - average: new element fits somewhere in the middle

```
insertSorted         :: Ord a => a -> [a] -> [a]
insertSorted e []     =  [e]
insertSorted e (x:xs)
  | e < x       = e : x : xs
  | otherwise   = x : insertSorted e xs
```

Slide 14

```
insertSorted         :: Ord a => a -> [a] -> [a]
insertSorted e []     =  [e]
insertSorted e (x:xs)
  | e < x       = e : x : xs
  | otherwise   = x : insertSorted e xs
```

Three functions approximate running time of `insertSorted`:

① $T^{best}_{insertSorted}$: in the best case, only three two are necessary (function call plus boolen operation)

$$T^{best}_{insertSorted}(n) = 2$$

② $T^{worst}_{insertSorted}$: n recursive calls, each requiring two steps

$$T^{worst}_{insertSorted}(n) = 2 * n$$

③ $T^{av}_{insertSorted}$: $n/2$ recursive calls, each requiring two steps

$$T^{av}_{insertSorted}(n) = n$$

Slide 15

Now running times T_{isort} of `isort`:

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

Best Case (what is it?):

- $n = 0$, only a single step is necessary
- otherwise, $T_{isort}(n)$ is the cost of
  ① applying `isort` to the tail (length $n - 1$) plus
  ② applying `insertSorted` to sorted tail

To summarise, we have a recursive equation

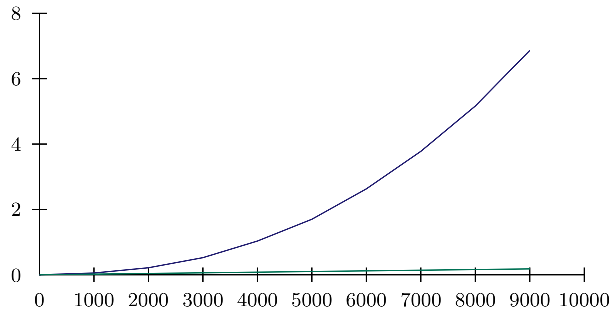
$$T^{best}_{isort}(0) = 1$$

$$T^{best}_{isort}(n) = T^{best}_{isort}(n-1) + T^{best}_{insertSorted}(n-1) + 1 = 3 + T^{best}_{isort}(n-1)$$

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

Slide 16

Running Times of `isort` on a randomly generated list and on a sorted list:



- For a sorted list, the costs of `isort` seem to grow linearly!

Slide 17

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

Worst Case (what is it?):

$$
\begin{aligned}
T^{worst}_{isort}(0) &= 1\\
T^{worst}_{isort}(n) &= 1 + T^{worst}_{isort}(n-1) + T^{worst}_{insertSorted}(n-1)\\
&= 1 + 2*(n-1) + T^{worst}_{isort}(n-1)\\
&= 2*n - 1 + T^{worst}_{isort}(n-1)
\end{aligned}
$$

What is the closed form of this recursive equation?

$$
\begin{aligned}
T^{worst}_{isort}(n) &= 1 + 2 + 4 + \ldots + 2*n\\
&= 1 + \sum_{i=1}^{n}(2*i)
\end{aligned}
$$

Slide 18

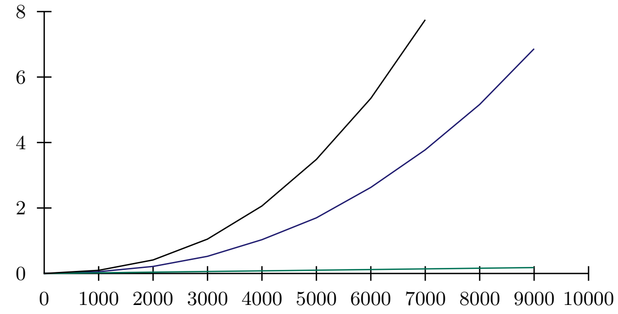
We know that

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

$$
\begin{aligned}
T^{worst}_{isort}(n) &= 1 + 2 + 4 + \ldots + 2*n\\
&= 1 + \sum_{i=1}^{n}(2*i)\\
&= 1 + 2*\sum_{i=1}^{n} i\\
&= 1 + 2*(n*(n+1)/2)\\
&= 1 + n*(n+1)\\
&= n^2 + n + 1
\end{aligned}
$$

Slide 19

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



Slide 20

On Average:

$$
\begin{aligned}
T^{av}_{isort}(0) &= 1\\
T^{av}_{isort}(n) &= 1 + T^{av}_{isort}(n-1) + T^{av}_{insertSorted}(n-1)\\
&= 1 + (n-1) + T^{av}_{isort}(n-1)\\
&= n + T^{av}_{isort}(n-1)\\
&\Rightarrow\\
T^{av}_{isort}(n) &= 1 + 2 + 3 + \ldots + n\\
&= \sum_{i=1}^{n} i\\
&= 0.5*n^2 + 0.5*n
\end{aligned}
$$

MORE EXAMPLES:

Slide 21

```
tail:: [a] -> [a]
tail (x:xs) = xs
```

- ➜ evaluation of `tail xs` requires a single step!

Slide 22

```
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

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

Slide 23

```
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 24

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
```

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

Slide 25

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

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

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

Slide 26

Summary

For `isort` we obtained the following results:

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

- ➜ This matches with our observations on the measured running times.
- ➜ Is this level of detail really needed?
- ➜ How careful must the set of basic steps be defined and the exact number of reduction steps be counted?

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