School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Complexity – Other notations than Big-O

Overview

- Summary on Linked lists
- Continue with the topic of complexity
  - More formal definitions and notations!

Summary on Linked lists

- We learned situations where array is not a good choice for representing lists
- We defined linked lists and learned a few methods for doing operations on linked lists
- Arrays:
  - Add at the end if enough space: O(1)
  - Due to fixed size, adding a new item at the end may take O(n)
  - Direct access to items by index number :O(1)
  - Shifts data when an item is added in the middle of the list or deleted from it: O(n)
- Linked Lists:
  - Dynamically grows or shrinks: add and remove take O(1)
  - No direct access by index number; Links should be followed: O(n)
  - Adding and removing items from the middle of the list include search: O(n), but not as costly as shifting the data

Review on Big O

- Big O is an upper bound on complexity

- How to find out if f(n) is in O(g(n))
  - Formal definition
  - lim f(n)/g(n) < ∞
  - There exists positive c and n_0 such that:
    - $f(n) <= c*g(n)$ for all $n>=n_0$
  - With some practice you will be able to tell this without much effort. But if we asked you for a proof, then go with the formal definition.

- Log
  - $\log n < \log^2 n < \log^3 n < .... < n^{0.001} < n^{0.01} < n^{0.1} < n < n \log n$
  - How about $\log n^2$?

Big Omega [$\Omega(g(n))$]

- $f(n) = \Omega(g(n))$ if there exist positive constants c and n_o such that $f(n) >= c*g(n)$ when $n >= n_o$.

- Can we say if $f(n)= O(g(n))$ then $g(n)= \Omega(f(n))$?

- We represent lower bounds with Big Omega

- Examples:
  - $0.5*n= \Omega(n)$?
  - $n^2= \Omega(n)$?
  - $\log n = \Omega(n)$?

Big Theta [$\Theta(g(n))$]

- $f(n) = \Theta(g(n))$ if and only if
  $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

- This is the tight bound.

- Examples:
  - $2n = \Theta(n)$?
  - $n \log n = \Theta(n)$?
  - $\log n = \Theta(n)$?

General Rules

- Some mathematical background is required for analyzing computational complexity

Rule 1. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then

1. $f_1(n) + f_2(n) =$
   $$O(g_1(n)+g_2(n))= O(\max(g_1(n),g_2(n)))$$
2. $f_1(n) * f_2(n) =$
   $$O(g_1(n)*g_2(n)))$$

Rule 2. If f(n) is a polynomial of degree k, then

$$f(n) = \Theta(n^k)$$

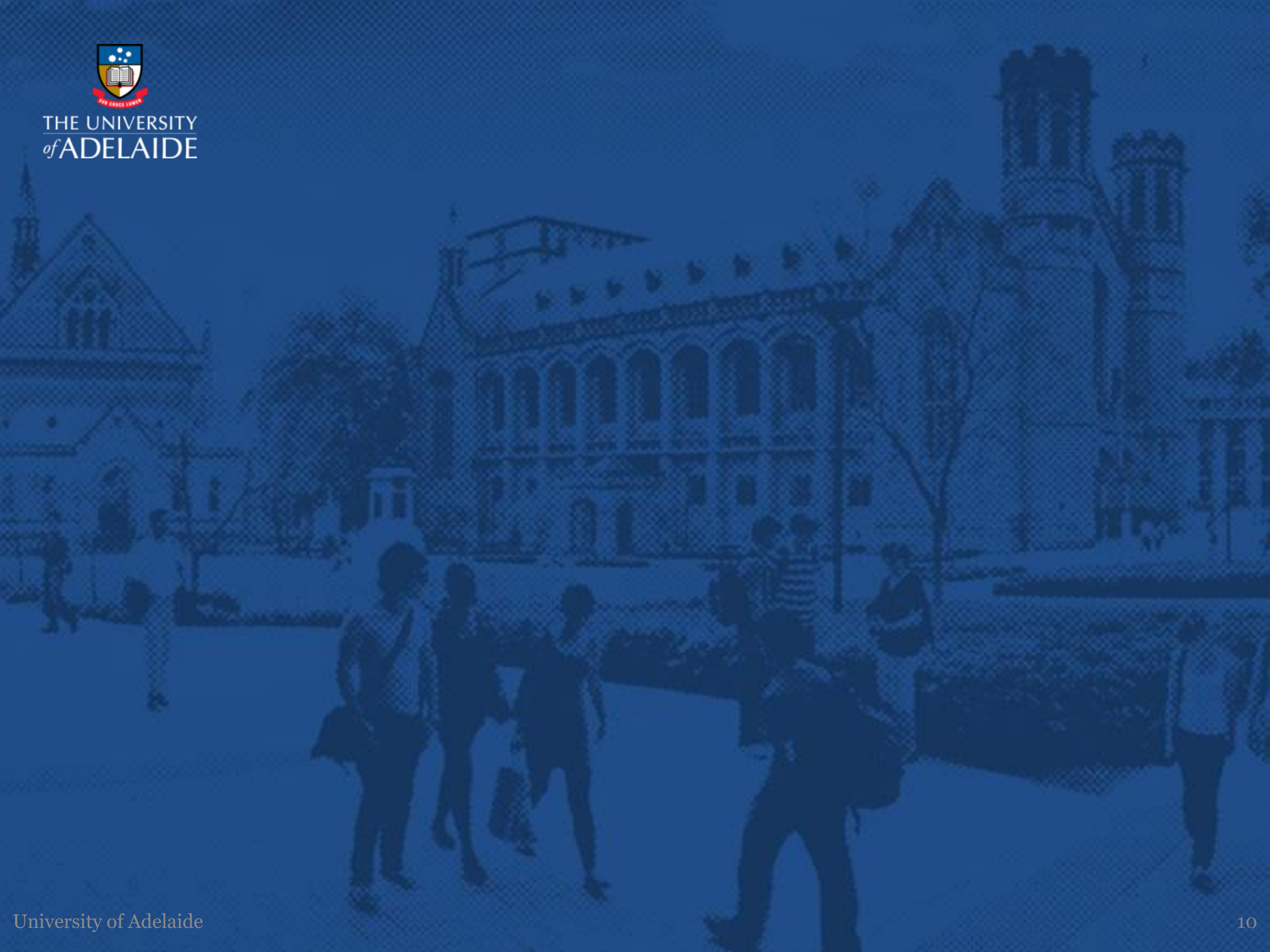
Rule 3. if $f(n)=n^{(1/k)}$ then $f(n)= \Omega(\log n)$ for any constant k.

Little o [o(g(n))]

- f(n) = o(g(n)) if for ***all*** constants c there exists an n_0 such that **f(n) < c*g(n)** when $n>n_0$.
  - In other words, lim (f(n)/g(n))=0 when n goes towards infinity
- Example:
  - n=o(n)?
  - $n=o(n^2)$?
- If f(n) = o(g(n)) as n -> infinity, then g(n) is growing much much faster than f(n).
  - The growth of f(n) is nothing when you compare it to g(n)
- Can we say if f(n)=o(g(n)) then f(n)=O(g(n))?
- Don't confuse big-Oh and little-oh
  - Big-Oh allows the possibility of the same growth rate.

Summary on these notations

- Big-Oh: $f(n)=O(g(n))$
  - Means $f(n)$ is bounded ABOVE by $g(n)$
- Big Omega (Ω) : $f(n)= \Omega(g(n))$
  - Means $f(n)$ is bounded BELOW by $g(n)$
- Big Theta (Θ): $f(n)=O(g(n))$, $f(n)= \Omega(g(n))$
  - Means $f(n)$ is bounded above and below by $g(n)$.
  - $g(n)$ is a tight upper and lower bound. It's hard to find.
  - Polynomials with degree k: $O(n^k)$ and $\Omega(n^k)$ => $\Theta(n^k)$
- Little o: $f(n)=o(g(n))$
  - Gives an upper bound
  - Stronger than Big O ($g(n)$ grows much faster than $f(n)$)
  - Does not allow the possibility of the same growth rate


THE UNIVERSITY
of ADELAIDE