School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Complexity

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
  - Do we need a destructor? How do you copy a linked list?

Review on Big O

- How to find out if f(n) is in O(g(n))
  - Formal definition
  - lim f(n)/g(n) =c, c>=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:
  - $0.5*n= \Omega(n)$?
  - $n^2= \Omega(n)$?
  - $\text{Log } n = \Omega(n)$?
    - NO

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:
  - $2n = \Theta(n)$?
  - $n \log n = \Theta(n)$?
    - No
  - $\log n = \Theta(n)$?
    - No

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. $f_1(n) * f_2(n) =$

   $O(g_1(n)*g_2(n)))$

Does this hold for Big Omega as well?

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)= Ω(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_o such that **f(n) < c*g(n)** when $n>n_o$.
  - In other words, lim (f(n)/g(n))=0 when n goes towards infinity
- Example:
  - n=o(n)?
    - No
  - 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): $f(n)= \Omega(g(n))$
  - 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

Simple Statement

- Simple statements:
  - Math operators: +, -, &&, *, etc …
  - Assignment , array indexing, …
  - Comparison

- The simple statements are all O(1)

- What about blocks of simple statements?

Simple Statement

- Consider the code block below

```
int next, n1, n2;

next = n1 + n2;
n2 = n1;
n1 = next;
```

- These statements are all O(1).
- They take a constant amount of time to execute, ***independent of the input size!***
- The complexity of the entire code segment is O(1).
  - These statements altogether still take a constant amount of time to execute.

Loops

- For-loops: (n is the input)

```
int counter = 0;
for(int i = 0; i< n ; i++){
  counter += i;
 }
```

- The running time of a for loop is at most the running time of the statements inside the for loop (including tests) multiplies the number of iterations.
- O(n*[complexity of statements inside the loop])
  - O(n)

Loops

```
int counter = 0;

for(int i = 0; i< 100; i++){

  counter += i;

 }
```

- The statements are performed 100 times.
- The complexity of the entire code segment is 100 * [the complexity of the statements inside the loop]
  - 100*c =O(1)

Loops

- Nested loops:

```
int counter = 0;

for(int i = 0; i< n ; i++){
  for(int j = 0; j< n ; j++){
    counter ++;
  }
 }
```

- The total running time of the statements that form a group of nested loops is the running time of the inner statements multiplied by the product of the sizes of all the loops.
- O(n^2)

Loops

- Nested loops

```
for(int i = 0; i< n; i++){
  for(int j = 0; j< m; j++){
    counter ++;
  }
}
```

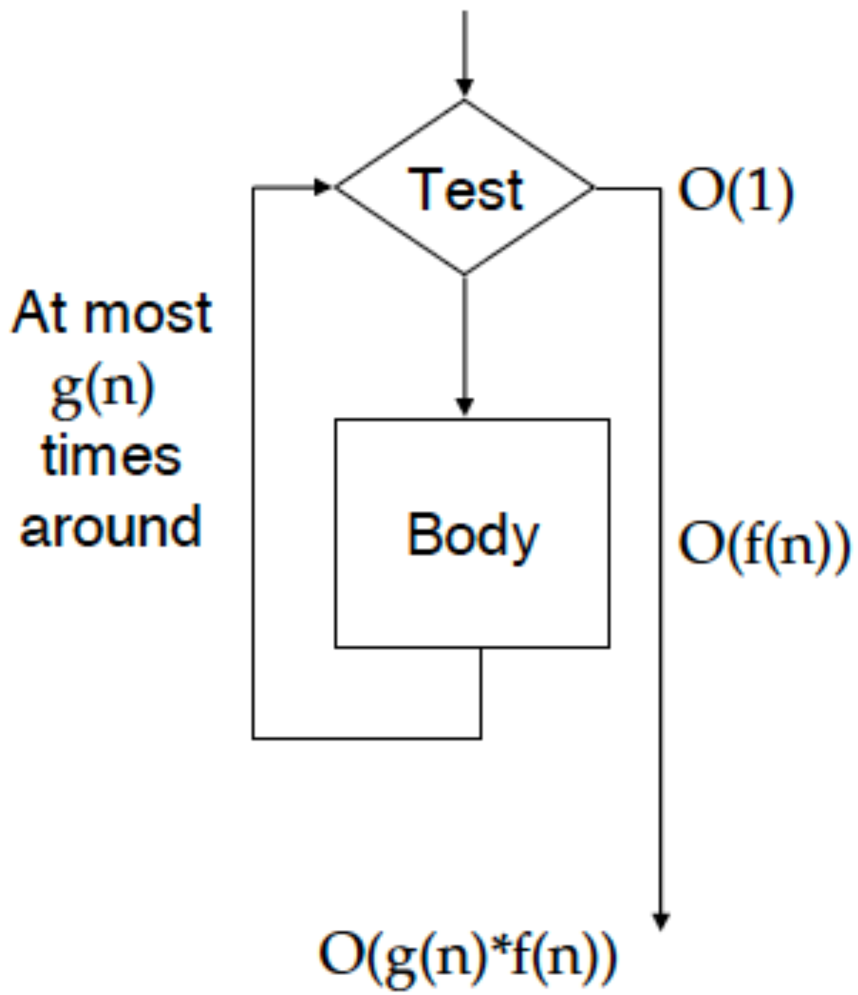
O(mn)

```
for(int i = 0; i< n; i++){
  for(int j = 0; j< 100; j++){
    counter ++;
  }
}
```

O(n)

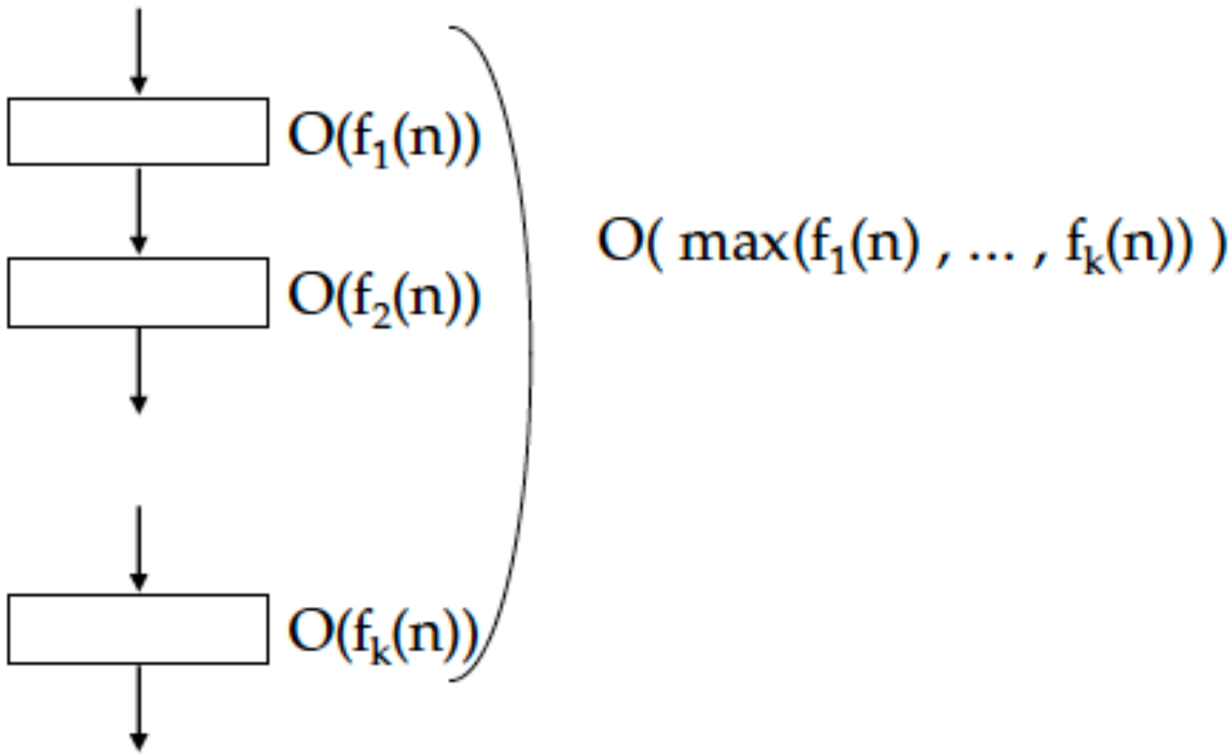
Loops

- While-loops:

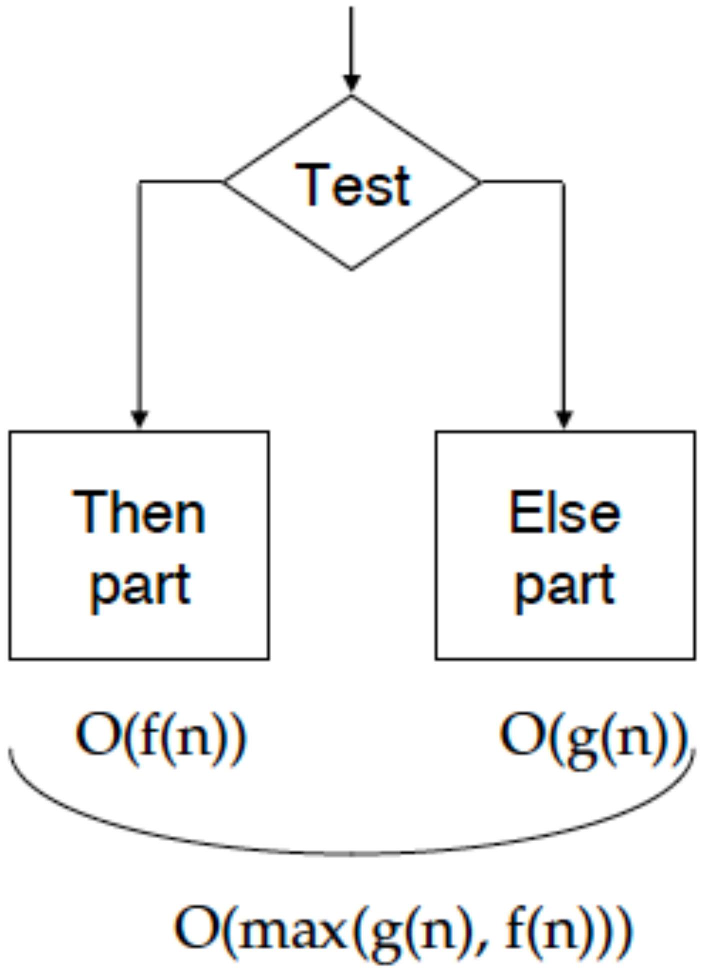


Consecutive Statements

- Block of statements without function calls is just summation.



If/else Statement



- The running time of an if/else statement is never more than the running time of the test plus the larger of the running times of the statements in the if and else block.

If/else Statement

```
if(a>b){
  for(int i=0; i<n; i++){
    counter ++;
  }
 }else{
  counter = 0;
 }
```

O(n)

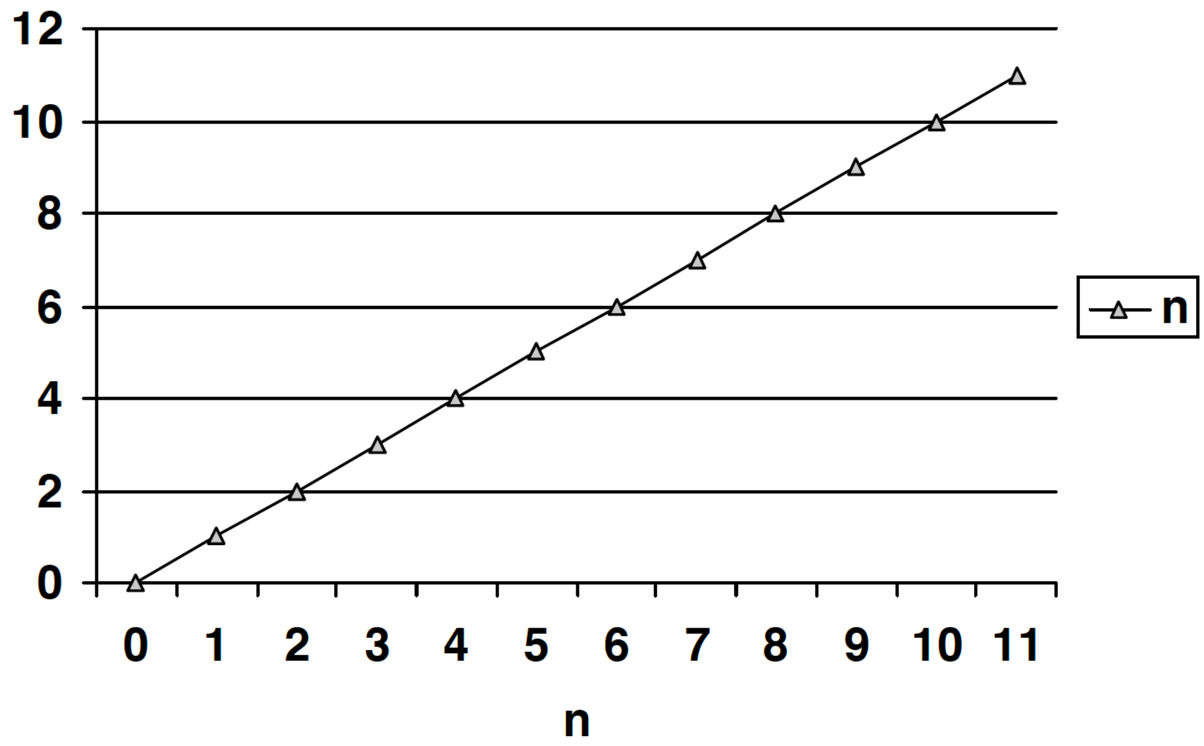
Example

- Iterative version of Fibonacci number calculation
- The program structure tree
- O(n)

```
int Fib(int n){
  if(n<0){
    return -1;
  }

  if(n == 0){
    return 0;
  }

  if(n == 1){
    return 1;
  }

  int n1 = 0;
  int n2 = 1;
  int next = 2;

  for(int current = 2; current <= n; current ++){
    next = n1 + n2;
    n2 = n1;
    n1 = next;
  }

  return next;
}
```

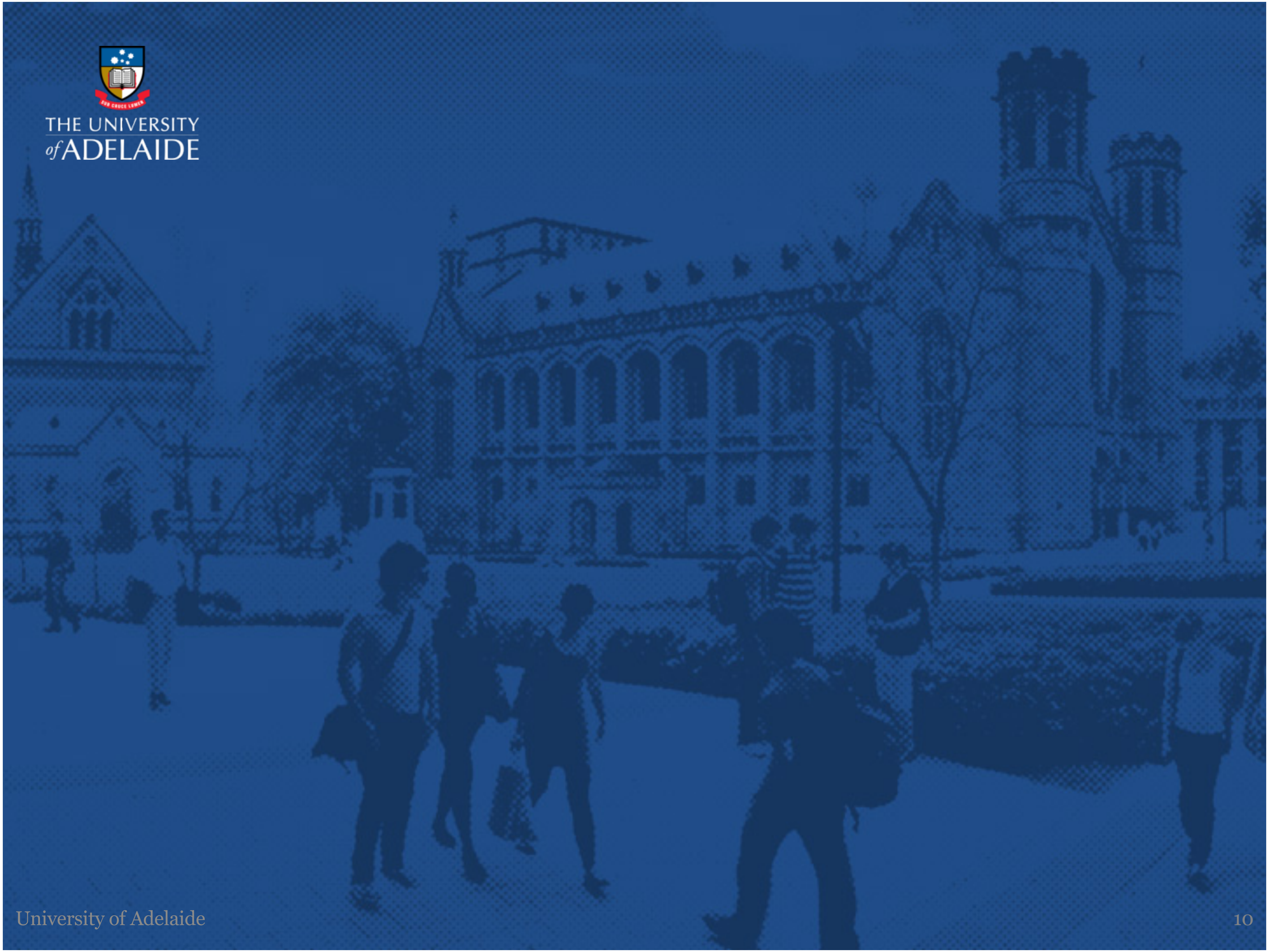
Example

- The iterative version of Fibonacci has a linear growth rate.
- The run time grows in proportion to the magnitude of the Fibonacci number we are computing.



Summary

- Notations:
  - Big O: for presenting an upper bound
- Simple Rules
  - Summation and multiplication
  - Polynomials with degree k: $O(n^k)$
  - Analysis of Simple algorithms:
    - Simple statement, If/else statements, Loops, Consecutive statement


THE UNIVERSITY
of ADELAIDE