School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Recursion 2

Review

- Recursion
- If a problem can be solved recursively, we can always do that iteratively as well.
- Sometimes, from a design or conceptual view, recursion is superior.
- There will always be efficiency issues in using recursion.

Overview

- In this lecture we will discuss:
  - More details of recursion.
  - Think recursively
  - How to improve the efficiency of recursion
    - Tail recursion
    - Helper function

Checklist for Recursion

- Three properties to be checked for a recursive algorithm:
  - There is no infinite recursion
  - Each stopping case performs the correct action
  - For all recursive cases: IF all recursive calls perform correctly, THEN the entire case performs correctly.
- To use recursion, break the problem into subproblems.
  - at least one sub-problem needs to be the same problem as the main problem, but with a smaller size (simpler).
- Why does this not work?

```
int countRow(Person i){
  if Person i is in the leftmost seat ||  Person i is in the rightmost seat
            return 1;
  else
    return countRow(Person i+1)+countRow(Person i-1);
}
```

Checking n!

- Infinite recursion?
  - The relationship n! = n * ((n-1)!) has n getting smaller and heading towards 0. Provided we have a base case for 0, no infinite recursion.
- Is the returned value for base correct?
  - Base case: n = 0, we return 1. This is correct.
- Is the relationship for recursion correct?
  - If (n-1)! returns the correct result, then n! = n * (n-1)! will return the correct result.
- Therefore, this is correct! (But what if n < 0?)

Think Recursively

- Many of the problems we talked before can be solved using recursion if we think recursively.

- Consider the palindrome problem in prac 1.

Problem Solving with Recursion

- Practice makes perfect!

- Step 1. Consider various ways to simplify inputs
  - Find sub-problems that perform the same task as the original problem, but with a simpler (or smaller) input.
- Step 2. Combine solutions with simpler inputs into a solution of the original problem.
- Step 3. Find solutions to the simplest cases.
- Step 4. Implement the solution by combining the simple cases and the reduction step.

Example

```
bool isPalindrome(string s){
        //base
        if(s.length()<=1)
                return true;

        //recursion
        if(tolower(s[0]) != tolower(s[s.length()-1]))
                return false;
        return isPalindrome(s.substr(1,s.length()-2));
}
```

Recursive Helper Functions

- This implementation of isPalindrome() is not efficient. Why?
  - It creates a new string for every recursive call
  - What about checking whether a substring is a palindrome or not?

- It is a common design technique in recursive programming to declare a second function that receives additional parameters.

```
int isPalindrome(string s, int start, int end)
```

Example

```
bool isPalindrome(string s){
        isPalindromHelper(s, 0, s.length-1);
}

bool isPalindromeHelper(string s, int start, int end){
        //base
        if(end==-1 || start=end)
                return true;

        //recursion
        if(tolower(s[start]) != tolower(s[end]))
                return false;
        return isPalindromeHelper(s,start+1, end-1);
}
```

No string created here. Reusing s. In previous solution, use of substr() created and returned another string.

Stack use for recursive isPalindrome

- How does it work?

Heads and Tails

- Some compilers perform optimisation to reduce your call overhead.
- If possible, the compiler will remove stack heavy operations and replace them with lighter ones.
- It's possible to do this, in recursion, using a technique called 'tail recursion'.

- A recursive function is tail recursive when recursive call is the last thing executed by the function.

Tail-recursive factorial

- How is stack used for this one?
- Here's a recursive factorial that a compiler can optimise to reduce stack overhead:

```
int fac(int n){
  if(n < 1){
    return 1;
  }else{
    return n*fac(n-1);
  }
}
```

```
int fac(int n, int acc){
  if (n < 1){
    return acc;
  }else{
    return fac(n-1,acc*n);
  }
}
```

More than one call

- Recursive functions can generate more than one call on each pass.
- Consider the Truckloads problem,
  - numTrucks(pile1, loadSize)+numTrucks(pile2, loadSize)

- Each call potentially generates two more calls
- Explosion!!
  - Exponential growth!
- You can see how this can quickly add up in terms of space

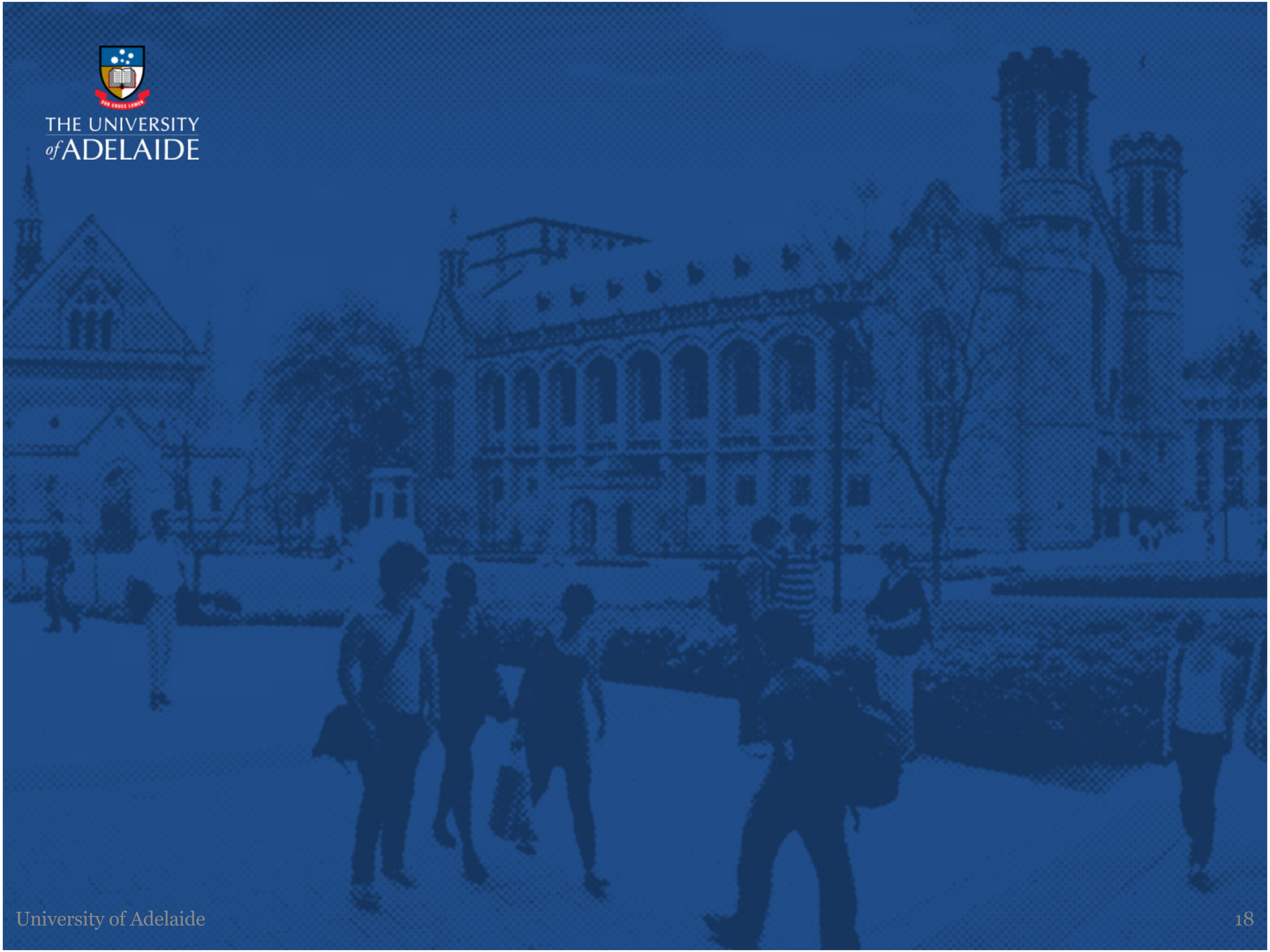
Hanoi Tower

- The Towers of Hanoi problem can be solved easily using recursion, but is difficult to solve without using recursion.

- The problem involves moving a specified number of disks of distinct sizes from one tower to another while observing the following rules:
  - Only one more tower can be used other these two towers
  - No disk can be on top of a smaller disk at any time
  - All the disks are initially placed on one tower
  - Only one disk can be moved at a time and it must be the top disk on the tower.

Summary

- Recursion is a useful tool for understanding problems and producing solutions, but:
  - You can always solve it iteratively
  - It can be inefficient and space hungry
  - Analysing recursive code can get tricky quickly


THE UNIVERSITY
of ADELAIDE