RECURSION

CITS1001
Scope of this lecture

- A brief introduction to the concept of recursion
- Simple recursive programs
- QuickSort

- This lecture is based on notes by Gordon Royle, UWA
Recursion

- We have already seen that a method in a class can call other methods, either in the same or other classes.
- However, a method in a class can also call itself.
- This self-referential behaviour is known as recursion.
- Recursion is an extremely powerful technique for expressing certain complex programming tasks, as it provides a very natural way to decompose problems.
- Despite this, there are costs associated with recursion and the careful programmer will always be aware of these.
The simplest example

• Consider the problem of computing the factorial function

\[ n! = n(n-1)(n-2)\ldots(1) \]

• So 2! = 2, 3! = 6, 4! = 24, 5! = 120, 6! = 720 and so on…

• However rather than define factorial in this way, we could give an equivalent, recursive definition

\[ 0! = 1 \]

\[ n! = n \times (n-1)! \]

• This definition is recursive because it uses factorial in the definition of factorial
Factorial in Java

```java
public long factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n*factorial(n-1);
}
```

We use `long` just to give the method slightly greater range.

The method `factorial` calls itself, but with a smaller argument.
What happens in the method call?

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * factorial(2)

factorial(2) = 2 * factorial(1)

factorial(1) = 1 * factorial(0)

factorial(0) = 1
Local variables

• Each recursive call to the method creates a “new copy” of the method – all the local variables are created again
• The compiler keeps track of them all so that the programmer cannot mix them up

\[
\text{factorial}(4) \text{ has a parameter } n \text{ equal to 4} \\
\quad \text{factorial}(3) \text{ has its own parameter } n, \text{ equal to 3} \\
\quad \quad \text{factorial}(2) \text{ has its own parameter } n, \text{ equal to 2} \\
\quad \quad \quad \text{factorial}(1) \text{ has its own parameter } n, \text{ equal to 1}
\]

• This means that you should *not* use recursion to replace straightforward iteration (like factorial), particularly if the loops are performed thousands or millions of times.
Ingredients for recursive definition

• Every valid recursive method definition needs two parts – the base case and the recursive part

• The *base case* gives the value that the method should return for some specified parameters
  • Usually the base case represents some sort of “trivial” case, such as a zero parameter or an empty list etc.

• The *recursive part* expresses the value to be returned in terms of another call to the same method, but with different parameters
  • To work correctly, the “different parameters” must be closer to the base case (in some sense)
Order is important

```java
public long factorial(int n) {
    return n*factorial(n-1);
    
    if (n == 0) {
        return 1;
    }
}
```

You must put the base case before the recursive part of the definition or bad things will happen – in this case a stack overflow will result!
More than one base case

- There is no need to have just one base case
- The (mathematical) definition of the Fibonacci numbers is a recursive definition with two base cases

\[
F_0 = 1 \\
F_1 = 1 \\
F_n = F_{n-1} + F_{n-2}
\]

- This gives the sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, …
Be careful though

```java
public int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    else {
        return fibonacci(n-1)+fibonacci(n-2);
    }
}
```

This looks fine, but in fact is disastrously slow because it unnecessarily repeats some calculations over and over again!
Binary Search

- Binary search can be expressed recursively in a very natural fashion, because we repeatedly perform the same operation of calculating the middle element and then searching in an array of half the size
- What is the “simplest case” in this situation?
- The length of the array is the parameter that is reduced at each stage of binary search and so the base case is when the left and the right bounds of the array are adjacent
- In this situation it is trivial to determine whether or not the element we are looking for is in the array or not
Binary Search

```java
boolean binarySearch(int[] a, int lf, int rt, int val) {
    if (rt – lf == 1)
        return a[lf] == val || a[rt] == val;

    int mid = (lf+rt)/2;
    if (a[mid] > val)
        return binarySearch(a,lf,mid,val);
    else
        return binarySearch(a,mid,rt,val);
}
```
Public interface

- This method would usually be made private
- The public interface would be the simpler

```java
public boolean binarySearch(int[] a, int val)
```

- Its only role would be to call the recursive version

```java
public boolean binarySearch(int[] a, int val) {
    return binarySearch(a, 0, a.length-1, val);
}
```
QUICKSORT
A recursive sorting algorithm

- Suppose you had to sort the following array with 16 elements

```
53 49 57 35 18 11 23 62 71 90 95 87 77 92 83
```

- Just before starting, you notice that the array has a very special structure

```
53 49 57 35 18 11 23 62
```

All smaller than 62

```
71 90 95 87 77 92 83
```

All larger than 62
Divide and conquer

• We can now divide the problem into two smaller problems

```
<table>
<thead>
<tr>
<th>53</th>
<th>49</th>
<th>57</th>
<th>35</th>
<th>18</th>
<th>11</th>
<th>23</th>
</tr>
</thead>
</table>
```
```
| 62 |
```
```
<table>
<thead>
<tr>
<th>71</th>
<th>90</th>
<th>95</th>
<th>87</th>
<th>77</th>
<th>92</th>
<th>83</th>
</tr>
</thead>
</table>
```
```
| 62 |
```
```
<table>
<thead>
<tr>
<th>11</th>
<th>18</th>
<th>23</th>
<th>35</th>
<th>49</th>
<th>53</th>
<th>57</th>
</tr>
</thead>
</table>
```
```
| 62 |
```
```
<table>
<thead>
<tr>
<th>71</th>
<th>77</th>
<th>83</th>
<th>87</th>
<th>90</th>
<th>92</th>
<th>95</th>
</tr>
</thead>
</table>
```
```
| 62 |
```
```

• The two “half-size” problems take much less than half the time!
But what if the array is not in this nice form?

Then we put it into this nice form!

First we choose an element to be the “fence”, and then we adjust the array so that the fence is in the correct position, everything to the left of the fence is smaller than it, and everything to the right of the fence is larger than it.

Choose the fence (we explain how later)
Find out-of-place elements

Now increase $L$ until reaching an element *bigger* than the fence

And decrease $R$ until reaching an element *smaller* than the fence
Then swap them over
Repeat

Move

\[
\begin{array}{cccccccccccccccc}
62 & 49 & 24 & 18 & 35 & 53 & 90 & 11 & 57 & 95 & 82 & 87 & 98 & 92 & 77 & 71 \\
\end{array}
\]

Exchange

\[
\begin{array}{cccccccccccccccc}
62 & 49 & 24 & 18 & 35 & 53 & 57 & 11 & 90 & 95 & 82 & 87 & 98 & 92 & 77 & 71 \\
\end{array}
\]

Until R < L
When $R < L$

When $R < L$, then every element to the right of $L$ is larger than the fence, and every element (except the fence) to the left of $L$ is less than the fence.

Swap the fence and element $R$.
public void partition(int[] a) {
    int fence = a[0];
    int left = 1;
    int right = a.length-1;
    while (right >= left) {
        while (left <= right && a[left] <= fence)
            left++;
        while (right >= left && a[right] >= fence)
            right--;
        if (right > left) {
            int swap = a[left];
            a[left] = a[right];
            a[right] = swap;
        }
    }
    a[0] = a[right];
    a[right] = fence;
}
public void partition(int[] a) {
    int fence = a[0];
    int left = 1;
    int right = a.length-1;

    while (right >= left) {
        while (left <= right && a[left] < fence)
            left++;
        while (right >= left && a[right] > fence)
            right--;

        if (right > left) {
            int swap = a[left];
            a[left] = a[right];
            a[right] = swap;
        }
    }

    a[0] = a[right];
    a[right] = fence;
}
public void partition(int[] a) {
    int fence = a[0];
    int left = 1;
    int right = a.length-1;
    while (right >= left) {
        while (left <= right && a[left] <= fence)
            left++;
        while (right >= left && a[right] >= fence)
            right--;
        if (right > left) {
            int swap = a[left];
            a[left] = a[right];
            a[right] = swap;
        }
    }
    a[0] = a[right];
    a[right] = fence;
}
QuickSort

- **QuickSort** is a recursive sorting method defined as follows:
- To sort an array,
  - Partition the array around some fence
  - **QuickSort** the elements of array *before* the fence position
  - **QuickSort** the elements of array *after* the fence position
- This is a recursive definition because we have used **QuickSort** in the *definition* of **QuickSort**
- We have only given the recursive part of the definition; what is the base case?
  - This is easy – if the array has 0 or 1 elements to be sorted, then we do not need to do anything!
In-place sorting

```java
private static int partition(int[] a, int start, int finish) {
    int fence = a[start];
    int left = start+1;
    int right = finish;

    // omitted code is identical to before

    a[start] = a[right];
    a[right] = fence;

    return right;
}
```

We just treat the elements `a[start]..a[finish]` as the array to be partitioned, and ignore the rest.

The method returns the correct position of the fence after it has been located.
The actual quickSort code

private static void quickSort(int[] a, int start, int finish) {
    if (finish-start > 0) {
        int fence_position = partition(a,start,finish);
        quickSort(a,start,fence_position-1);
        quickSort(a,fence_position+1,finish);
    }
}

public static void quickSort(int[] a) {
    quickSort(a,0,a.length-1);
}
The call `quickSort(a)`

First calls `partition(a, 0, 15)`, which changes `a` to

and returns 7 as the fence position

`quickSort(a, 0, 6)` sorts the left half

`quickSort(a, 8, 15)` sorts the right half
Comments on QuickSort

• Works well when the fence position ends up being roughly halfway through the array, as the two sub-problems are then about equal in size

• Our choice of fence – position 0 – means that the worst case for QuickSort is when the array is already ordered

• In practice, it pays to avoid this case
  • Pick the fence randomly
  • Randomly permute the array before sorting with QuickSort
  • Sample a small portion of the array, and choose the median as fence

• Average number of comparisons is \( n \log_2 n \)
Java library

- The Java library includes a number of optimized methods for sorting.
- See, for example,
- Arrays.sort and
- Collections.sort(list)
Summary

• Recursion can be used to express some complex computations in an elegant way.
• However, there are overheads for using recursion that the programmer needs to be aware of.

• The Quicksort algorithm is efficient, O(n log n), because it breaks a large list into smaller ones and sorts those.
• However, Quicksort is not always the best solution. For example, it’s worst performance is when the list is already sorted.

• The Java library includes optimized methods for sorting. See, for example, Arrays.sort and Collections.sort(list)