Recursion

1 Factorials

The factorial of a positive integer n is the product of the integers 1 to n, hence we can implement the calculation of factorials using recursion

```
#Iterative Factorial
total=1
for i=1 to n do
    total=total×i
end for
return total

#Recursive Factorial factorial(n)
if n=0 then
    return 1
else
    return n×factorial(n-1)
end if
```

2 Recursive Algorithms

- A recursive algorithm must have a **base case**
- A recursive algorithm must **change** its state and move towards the base case
- A recursive algorithm must **call itself**, recursively

3 Lists

```
# Iterative sum of a list L
sum=0
for i in L do
    sum=sum+i
end for
return sum

#Recursive sum of a list L:listsum(L)
if len(L)=1 then
    return L(0)
else
    return L[0]+listsum(L[1:])
end if
return sum
```

4 Examples on the whiteboard

4.1 Fibonacci

```
#Iterative Fibonacci sequence
input n
fibs=[0,1]
for i=2 to n
    fibs=fibs+sum of last 2 entries
return fibs[n]
```

```
# Recursive Fibonacci sequence
Input n
for n ⩽ 1
    return n
fibs
    return fibs(n-1)+fibs(n-2)
```

- This is exceptionally slow as the whole previous Fibonacci sequence is calculated in order to calculate each number, whereas with iteration the values are calculated then stored.
- The number of calls will be on the order of 2^N with recursion, whereas iteration will use N calls
- A way to speed up the recursive step is to store the values calculated, as values are calculated multiple times in the previous method

4.2 Palindromes

```
input string s
output true or false
isPalindrome(S)
    if length S⩽1 then
        return true
    end if
    if the first and last characters don't match then
        return false
    else
        isPalindrome(S with first and last characters removed)
```

4.3 Flood fill a bounded area

```
floodfill(x,y)
    input screen,pixel(x,y)
    output screen with the contiguous region that contains that pixel filled with the colour
    if (x,y) is filled then
        return
    fill(x,y) #change (x,y) to blue
    floodfill(x+1,y),floodfill(x-1,y),floodfill(x,y+1),floodfill(x,y-1)
```

5 Memoization

Store the obtained results.

For example a memoized recursive algorithm for fibonacci numbers would be:

```
fib_memo[0]=0
fib_memo[1]=1
fib(n) input n, output nth Fibonacci Number
if not fib_memo[n]
    fib_memo[n]=fib(n-1)+fib(n-2)
return fib_memo[n]
```

6 Backtracking

- A technique for problems with many **candidate** solutions but too many to try
- For example there are a huge number of ways to fill a sudoku grid
- General idea: Build up the solution one step at a time, **backtracking** when unable to continue

6.1 Informal generic algorithm

1. Do I have a solution yet?
2. No. Can I extend my solution by one "step"?
3. If yes, do that
4. Do I have a solution now? If yes, I'm done
5. If not, try and extend again
6. When I can't extend, take one step back and try a different way
7. If no other extension available, then give up - no solution can be found

6.2 Pseudocode generic algorithm

```
#extend_solution(current solution)
if current solution is valid then
    if current solution is complete then
        return current solution
    else
        for each extension of the current solution do
            extend_solution(extension)
        end for
    end if
end if
```

7 Knights tour

A knight's tour: To move a knight around a chessboard such that each square is visited exactly **once** before the knight returns to its **starting position**

```
#extend_solution(current solution)
if cnew move is to unvisited square on board then
    if every square is visited then
        return current solution
    else
        for each of eight possible moves do
            extend_solution(with move added)
        end for
    end if
end if
```

7.1 Implementing Knights Tour

- Rather than having a list of moves made, it is easier to maintain an 8×8 array recording when each square was visited (initially all values are zero)
- Use a counter to record how many squares have been visited
- If every square has been visited, we must also check that we can return to the start in one more move

8 Queens Problem

Queen's Problem: Find all possible ways of placing 8 queens on a chessboard such that none attacks any other

```
#extend_solution(current solution)
if new queen is not under attack then
    if 8 queens on board then
        return current solution
    else
        for each of eight squares in the next row do
            extend_solution(with extra queen)
        end for
    end if
end if
```