Welcome to COMP1081 - Algorithms and Data Structures

The lecture will begin at 5 past. While you wait, join respond to the PollEverywhere question by SMS or on the web at `https://pollev.com/eamonn`

Consider the sequence of Fibonacci numbers $F(n)$ defined by

$$F(n) = \begin{cases} 0 & \text{for } n = 0 \\ 1 & \text{for } n = 1 \\ F(n-1) + F(n-2) & \text{for } n \geq 2 \end{cases}$$

Calculate $F(10)$.

Topic 5: Recursion and Backtracking

Eamonn Bell

eamonn.bell@durham.ac.uk

Recursive Algorithms: Factorials

- The factorial of a positive integer n is the product of all integers from 1 to n, often denoted $n!$.
- For example, $5! = 5 \times 4 \times 3 \times 2 \times 1$

Recursive Algorithms: Factorials

- The factorial of a positive integer n is the product of all integers from 1 to n, often denoted $n!$.
- For example, $5! = 5 \times 4 \times 3 \times 2 \times 1$
- Observe that $5! = 5 \times 4!$

Recursive Algorithms: Factorials

- The factorial of a positive integer n is the product of all integers from 1 to n, often denoted $n!$.
- For example, $5! = 5 \times 4 \times 3 \times 2 \times 1$
- Observe that $5! = 5 \times 4!$
- In this way, the problem of computing 5! breaks down into two subproblems:
  - Computing 4!
  - ...and multiplying the result of that, by 5

Recursive Algorithms: Factorials

- The factorial of a positive integer n is the product of all integers from 1 to n, often denoted $n!$.
- For example, $5! = 5 \times 4 \times 3 \times 2 \times 1$
- Observe that $5! = 5 \times 4!$
- In this way, the problem of computing 5! breaks down into two subproblems:
  - Computing 4!
  - ...and multiplying the result of that, by 5
- In turn, the problem of computing 4! can be broken down in a similar way.

Recursive Algorithms: Factorials

- The factorial of a positive integer n is the product of all integers from 1 to n, often denoted $n!$.
- For example, $5! = 5 \times 4 \times 3 \times 2 \times 1$
- Observe that $5! = 5 \times 4!$
- In this way, the problem of computing 5! breaks down into two subproblems:
  - Computing 4!
  - ...and multiplying the result of that, by 5
- In turn, the problem of computing 4! can be broken down in a similar way.
- This makes the factorial function a candidate for implementation as a **recursive** algorithm.

Recursive Factorial: factorial(n)

```
if n=1 then
    return 1
else
    return n × factorial(n-1)
end if
```

Recursive Factorial: factorial(n)

```
if n=1 then
    return 1
else
    return n × factorial(n-1)
end if
```

Iterative Factorial

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

Recursive Algorithms

- A recursive algorithm is an algorithm that calls itself to do part of its work.
- A recursive algorithm must have a base case.
- A recursive algorithm must change its state and move toward the base case.
- A recursive algorithm must call itself, recursively.

Iterative Sum of a list L

```
sum = 0
for i in L do
    sum = sum + i
end for
return  sum
```

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
```

Memoization

- A recursive implementation of the function to compute the nth Fibonacci number will be called several times with the same argument.
- We could store the result of these intermediate function calls, because these results do not change over time.

Memoization

- A recursive implementation of the function to compute the nth Fibonacci number will be called several times with the same argument.
- We could store the result of these intermediate function calls, because these results do not change over time.
- **Storing the result of a computation so that it can be subsequently retrieved without repeating the computation** is called memoization.
- Hash tables are a good choice of data structure to implement memoization.

The lecture will begin at 5 past. While you wait, join respond to the PollEverywhere question by SMS or on the web at `https://pollev.com/eamonn`

Consider an implementation of the function **floodfill** given below, where *x* and *y* represent pixel locations in a 2D screen

floodfill(x, y)

```
if (x,y) is out of range or already filled then
    return
end if
colour in pixel at location (x,y)
floodfill(x+1,y)
floodfill(x-1,y)
floodfill(x,y+1)
floodfill(x,y-1)
```

A Recursive Technique: Backtracking

- A technique for problems with many candidate solutions but too many to try.
- For example: there are 6,670,903,752,021,072,936,960 ways to fill in a sudoku grid.

A Recursive Technique: Backtracking

- A technique for problems with many candidate solutions but too many to try.
- For example: there are 6,670,903,752,021,072,936,960 ways to fill in a sudoku grid.
- General idea: build up the solution one step at a time, backtracking when unable to continue.

Generic algorithm (informal)

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.

▸ Sudoku demo from Wikimedia Commons

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
```

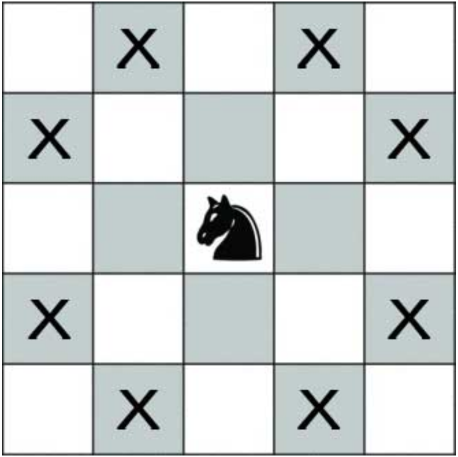
Generic algorithm

```
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
```

For sudoku, start by calling extend_solution with the partially filled grid that is given to you.

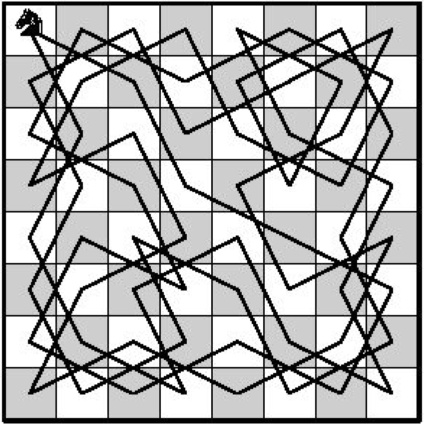
Knights

A knight is a chess piece that can move by moving one square in one direction and two squares in a perpendicular direction.



A Knight's Tour

A Knight's Tour: to move a knight around a chessboard such that each square is visited exactly once.



A Knight's Tour: using the generic algorithm

- What is a (partial) solution?

A Knight’s Tour: using the generic algorithm

- What is a (partial) solution?
  - Just a list of squares in the order they are visited.

A Knight's Tour: using the generic algorithm

- What is a (partial) solution?
  - Just a list of squares in the order they are visited.
- When is it valid?

A Knight's Tour: using the generic algorithm

- What is a (partial) solution?
  - Just a list of squares in the order they are visited.
- When is it valid?
  - No squares visited more than once.

A Knight's Tour: using the generic algorithm

- What is a (partial) solution?
  - Just a list of squares in the order they are visited.
- When is it valid?
  - No squares visited more than once.
  - Knight has not jumped off the board

A Knight's Tour: using the generic algorithm

- What is a (partial) solution?
  - Just a list of squares in the order they are visited.
- When is it valid?
  - No squares visited more than once.
  - Knight has not jumped off the board
- When is it complete?

A Knight's Tour: using the generic algorithm

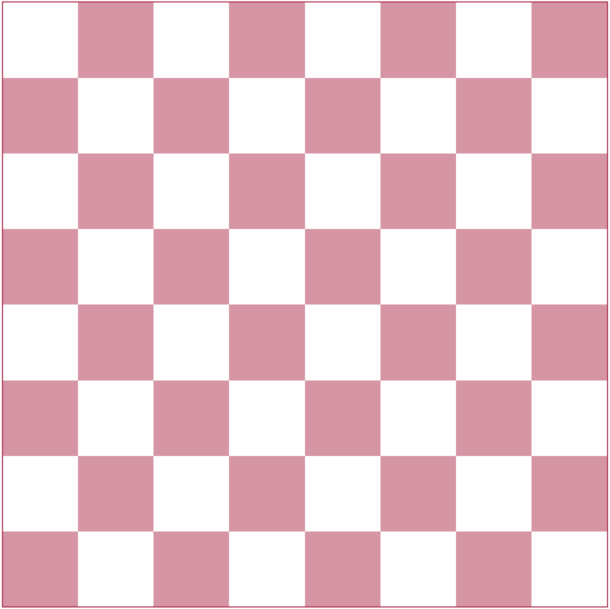
- What is a (partial) solution?
  - Just a list of squares in the order they are visited.
- When is it valid?
  - No squares visited more than once.
  - Knight has not jumped off the board
- When is it complete?
  - Every square visited: 64 items in the list.

A Knight's Tour: using the generic algorithm

- What is a (partial) solution?
  - Just a list of squares in the order they are visited.
- When is it valid?
  - No squares visited more than once.
  - Knight has not jumped off the board
- When is it complete?
  - Every square visited: 64 items in the list.
- How can the current solution be extended?

A Knight's Tour: using the generic algorithm

- What is a (partial) solution?
  - Just a list of squares in the order they are visited.
- When is it valid?
  - No squares visited more than once.
  - Knight has not jumped off the board
- When is it complete?
  - Every square visited: 64 items in the list.
- How can the current solution be extended?
  - Consider each of the eight possible moves.



Using the 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
```

Using the 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
```

Using the generic algorithm

extend_solution(current solution)

```
if current solution is valid then
    if current solution is complete then
        return current solution
    else
        for each of eight possible moves do
            extend_solution(extension)
        end for
    end if
end if
```

Using the generic algorithm

```
extend_solution(current solution)
  if current solution is valid then
      if current solution is complete then
          return current solution
      else
          for each of eight possible moves do
              extend_solution(current solution with move added)
          end for
      end if
  end if
```

Using the generic algorithm

```
extend_solution(current solution)
  if current solution is valid  then
      if current solution is complete  then
          return current solution
      else
          for each of eight possible moves do
              extend_solution(with move added)
          end for
      end if
  end if
```

Using the generic algorithm

```
extend_solution(current solution)
  if new move is to unvisited square on the board then
      if current solution is complete then
          return current solution
      else
          for each of eight possible moves do
              extend_solution(with move added)
          end for
      end if
  end if
```

Using the generic algorithm

```
extend_solution(current solution)
  if new move is to unvisited square on the board then
      if current solution is complete then
          return current solution
      else
          for each of eight possible moves do
              extend_solution(with move added)
          end for
      end if
  end if
```

Using the generic algorithm

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

Using the generic algorithm

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

So we have an algorithm for Knight's Tour . . .

Implementing Knight's 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).

Implementing Knight’s 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.

Implementing Knight's 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.

The algorithm is practical for a 6×6 board, but rather slow for an 8×8 board and impractical for much larger boards. What additional ideas could we add to the algorithm?