Recursion

"To iterate is human, to recurse divine."

(L. Peter Deutsch)

Mathematical Interlude: Recursion and Induction

Example: The Factorial Function

$$N! = FACT(N) = 1 \cdot 2 \cdot 3 \cdot ... \cdot N$$

Recursive Definition:

$$\begin{cases} FACT(0) = 1 \\ FACT(N) = N \cdot FACT(N-1) \end{cases}$$

Exercise:

Use the recursive definition to calculate FACT(5) = 5!

Mathematical Interlude: Recursion and Induction

Recursion:
A recursive definition of a function, involves the function itself.
In a programming language, a recursive routine or function is a function which calls itself.

Induction:
is a mathematical proof technique based on the properties of the natural numbers:

1) Assume true an initial “base” statement (example: 0! =1).

2) Assume the statement is true for N

3) Prove it is true also for N+1

Mathematical induction is strongly related to recursion.

Exercise

Prove the integers summation formula by induction.

$$S(N) = \sum_{i=1}^{i=N} i = \frac{N(N+1)}{2}$$

Analysis of the Recursive Factorial

```
void main(void) {
    int n;
    cin >> n;
    cout << factorial(n) << endl;
}

int factorial(int n) {
    int result;

    if(n==1) return 1;
    else result = n * factorial(n - 1);
    return result;
}
```

Base Case

In constructing a recursive solution, the base case is crucial for the end of the computation. The base case is explicitly stated in the mathematical definition (see slides before).

Another Example: The Fibonacci Sequence

$$\begin{cases} FIB(1) = 1 \\ FIB(2) = 1 \\ FIB(N) = FIB(N-1) + FIB(N-2) \end{cases}$$

c. 1170-1240

Exercise:

1) Try to use recursion “on paper”
2) Write a program implementing FIB by recursion
3) Calculate the (approximate) limit of the FIB(N)/FIB(N-1) succession
4) Would be an iterative solution better? If yes, why?

Another Example: Fermat's Infinite Descent

$$\begin{cases} ID(1) = 2 \\ ID(N) = 2 \cdot ID(N-1) \end{cases}$$

c. 1601-1665

This recursion enters in Fermat's proof for the number of subsets of a set of N elements.
Question: how many subsets the set A={1,2,3} has?

Exercise:

1) Can you find a non recursive form of the ID function? ID(N) = …
2) Write a program implementing ID.

Just for fun: Fermat's Proof:

Let S(n) be the number of subsets of a set of n elements.
Consider the element x and consider the subsets containing it or not.

There are S(n-1) subsets not containing x.
There are also S(n-1) subsets containing x (since you can just take the sets without x and add it to them!). Therefore:

$$S(n) = S(n-1) + S(n-1) = 2 \cdot S(n-1)$$

For a set with n-1 elements:

$$S(n-1) = 2 \cdot S(n-2)$$

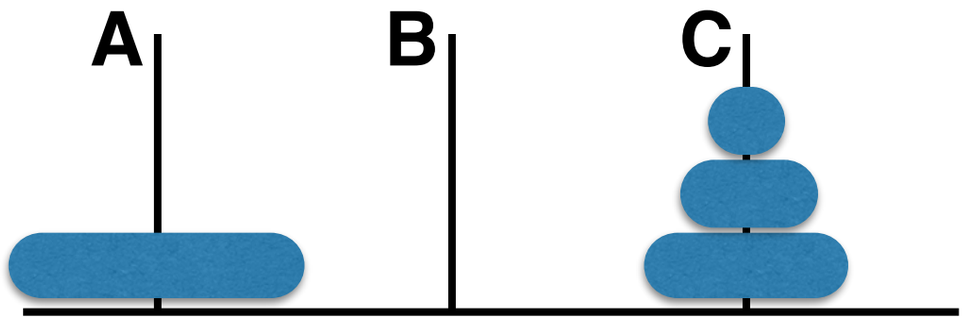
Substituting the second formula in the first up to the base case, it is easy to infer that:

$$S(n) = 2^n$$

A more complicated case: The Towers of Hanoi

The game consists to move all the disks from the first peg to the last. You can move only one disk at a time and smaller disks can only stay on top of larger disks.

The Towers of Hanoi: Analysis



Initial State: all disks on peg A
Simple case: 1 disk -> simply move it from A to C.
For N disks:

- Solve the problem for N-1 disks (move from A to C, B is the spare).
- After this, only the largest disk remains on A
- Move the large disk from A to B
- Move the N-1 disks from C to B
  this means to solve the problem with A as a spare this time.

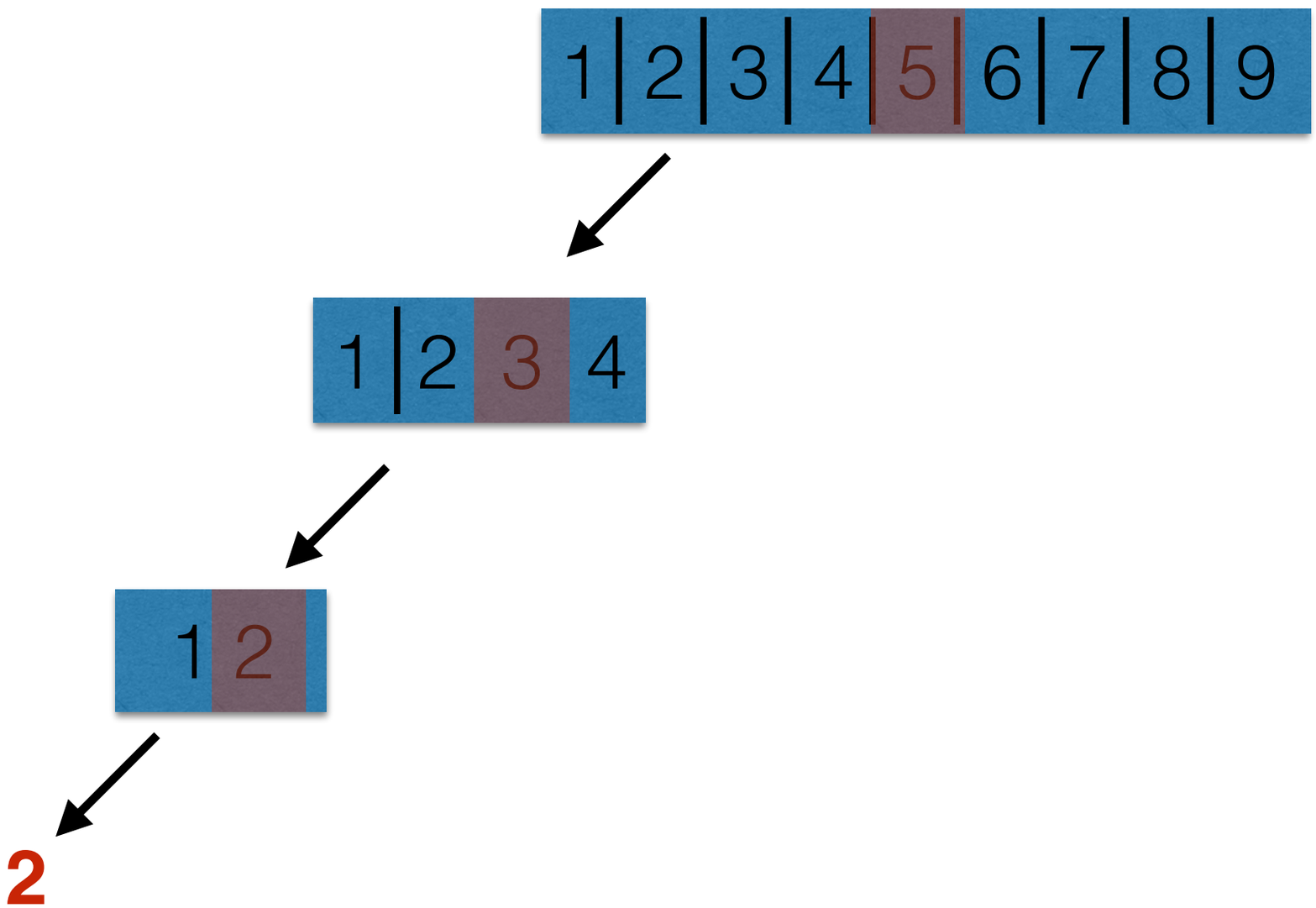
```
void solve(int count, char source, char dest, char spare){

  if (count == 1) cout << "Move from " << source << " to " << dest << endl;

  else {
    solve(count-1,source,spare,dest); //move n-1 disks from A to C
    solve(1,source,dest,spare);       //move the remaining disk from A to B
    solve(count-1,spare,dest,source); //move n-1 disks from C to B ...
  }
}
```

```
int main()
{
  solve(5,'A','B','C');
}
```

Binary Search

Search through an unordered structure is usually an O(N) process.
A way to speed up a search is to operate on ORDERED (sorted) structures.

EXAMPLE: Binary search on sorted arrays.
Search the number 2:



Binary Search: Recursion Solution

```
int binary_search(int array[], int left, int right ,int item)
{
  int middle;

  //middle element of the array
  middle = ( left + right ) / 2;

  //element is in the left half of the array
  if (array[middle] > item) return binary_search(array, left, middle - 1,item);

  //element is in the right half of the array
  else if (array[middle] < item) return binary_search(array, middle + 1, right,item);
  //the element has been found

  else return middle;
}
```

Binary Search: Iterative Solution

```
int binary_search( int array[], int numItems, int item )
{
  int left = 0;
  int right = numItems - 1;
  int middle;
  int found = FALSE;
  int location = -1;

  while( left <= right && !found )
    {

      //middle element of the array
      middle = ( left + right ) / 2;

      // element is in the left half of the array
      if( item < array[middle])
        {
          right = middle - 1;
   }
      // element is in right half of thr array
      else if( item > array[ middle ] )
        {
          left = middle + 1;
   }
      // the element has been found
      else
        {
          found = TRUE;
          location = middle;
        }
    }
  return location;
}
```

Sometimes the recursive solution code is much clearer (and shorter!) than the corresponding iterative one.

Recursion vs Iteration

Is one technique better than the other?
Depends…

Use a recursive solution if:

- There is a clear advantage
- The problem/code is way clearer/shorter

Do not use recursion if:

- The iterative solution is much simpler
- Memory requirements favor iteration

And in general:

- It depends very much on the structure of the problem.

Recursion vs Iteration

Example: The factorial function
The factorial is a typical example where the iterative solution is in principle better, since for calculating F(N) you need F(N-1) which is already available as result of the previous iteration, without the need of memorizing all the intermediate steps.

Another Example: The Fibonacci sequence
For calculating F(N) you need F(N-1) and F(N-2): with an iterative solution you just memorize two previous results and not the full sequence.

Binary Search:
This is one example where the recursive solution is clearer than the iterative. We will see in the near future that recursive searches are very convenient in more complex data structures.

The Tower of Hanoi:
In this case, it is pretty clear that the recursive solution is superior from the point of view of :

- Clarity
- Implementation (short code)

From Recursion to Iteration, and back.

Question:
Are recursion and iteration equivalent? I.e.: is it always possible to turn a recursive routine into an iterative one?

Answer: YES!

Simplest Case: Tail-Recursion.
A tail-recursive function is a function which ends by ONLY calling itself.
The factorial is not a tail-recursive function, since it calls itself TIMES n.
In the case of tail-recursive functions, the translation to iterative code is straightforward:

More efficient!

```
f (x){
   if (<condition>) return f(x);
   else return g(x);
}
```

→

```
   f(x){
      while (condition)
         x = k(x);
      return g(x);
   }
```

In-class exercise

Implement a recursive version of the toVector function for the LinkedList ADT:

```
void LinkedBag<T>::toVector2(vector<T>& content, Node<T>* curPtr) const
```

See Pg.148 in the book.
The idea is to create a recursive “fillVector” function:

```
void LinkedBag<T>::fillVector(vector<T>& bagContents, Node<T>* curPtr)
```

which calls itself but referencing to the next pointer at each iteration:

```
bagContents.push_back(curPtr->getItem());
fillVector(bagContents,curPtr->getNext());
```

In-class exercise

Write a recursive algorithm which, given a vector, returns it in reverse order.

An important strategy: BackTracking

Backtracking is useful an useful **recursive** algorithms for solving **constrained satisfaction** problems.
The algorithm:

- Builds incrementally on candidate solutions.
- "Backtracks" when it is clear that the final solution cannot be reached.

Example problems:

- The "8 Queens Problem"
- Finding the way out from a maze
- Solving crossword or sudoku puzzels
- In general, constrained optimization problems

A variant of the "maze" problem is proposed in the book:

- Find a possible flight route between two cities.
  - The constraints are:
    The start-end cities
    The allowed paths are the only ones where there is a flight available.


ITALY
destination
origin
Liechtenstein
Switzerland
Austria
Hungary
Slovenia
Croatia
Bosnia and Herzegovina
Serbia & Montenegro
France
Monaco
San Marino
Vipiteno
Bórmio
Tolmezzo
Sondrio
Trento
Belluno
Verbania
Aosta
VALLE D'AOSTA
LOMBARDY
TRENTINO-ALTO ADIGE
FRIULI-VENEZIA GIULIA
VENETO
Trieste
Novara
Milan
Venice
Torino
Piacenza
Mantova
Rubano
PIEMONTE
Ferrara
Cuneo
Genoa
LIGURIA
Ravenna
Bologna
Massa
La Spézia
Imperia
Pisa
Fano
Ancona
Ligurian Sea
Urbánia
TOSCANA
MARCHE
Montepulciano
Perúgia
Ascoli Piceno
Adriatic Sea
Elba
Grosseto
UMBRIA
Aquila
Viterbo
Vasto
Vieste
Roma
Vatican City
Frosinone
MOLISE
LAZIA
Bari
Tempio Pausánia
Palau
Napoli
Avellino
Fasano
PUGLIA
Brindisi
Sassari
Olbia
Potenza
CAMPANIA
BASILICATA
Lecce
Alghero
Nuoro
Táranto
Macomer
Sardinia
Rofrano
SARDEGNA
Gulf of Taranto
Ugento
Scalea
Sanluri
Tyrrhenian Sea
Torretta
San Vito
CALABRIA
Carbonia
Cagliari
Isole Eolie
Catanzaro
Ionian Sea
Mediterranean Sea
Palermo
Messina
Siderno
Trápani
Reggio di Calabria
Marsala
Salemi
SICILIA
Catánia
Strait of Sicily
Agrigento
Siracusa
Caltagirone
Sicily
Pantelleria
Ragusa
Mediterranean Sea
Algeria
Tunisia
Isole Pelagie
Malta
Not to Scale
Lampedusa
©ephotopix.com
ephotopix

Find a Flight Path with Backtracking

```
bool isPath(City origin , City destination){
   bool result, done;
   markVisited(origin);

   if (origin == destination) result = true;
   else {
      done = false;
      City next = getNext(origin);

      while (!done && next != NO_CITY){
         done = isPath(next,destination);
         if (!done) next = getNext(origin);
      }

   }

   return result;
}
```

Where is the backtracking code?