Recursion

Recursion is a technique which is useful when a task can be split into similar, but smaller, subtasks.

When an algorithm calls itself, we say that the algorithm is RECURSIVE.

Let's think about the algorithm for calculating the power n of a natural number x: power(x,n)

e.g. power(3,2) = 8 $\Rightarrow$ $2^3 = x^n$

We can have an ITERATIVE algorithm:

```
POWER(x, n)
    result = 1
    for i=1 to n do
        result = result * x      ⤳ result *= x
    return result
```

RECURSIVE version:

```
POWER(x, n)
    if (n==1) then      ⤳ BASE of RECURSION because it produces the obvious result pow(x,1)=x
        return x
    else
                          recursive call
        return x * pow(x, n-1)      ⤳ RECURSIVE STEP ⤳ this goes on until n reaches 1
                                                                                  base case
```

The execution stack ⤳ a stack is a LIFO structure

last in first out

When an algorithm calls itself (nested call):

1) the current execution is paused
2) the execution context associated with the current algo-exec is stored in a stack ⤳ local variables
3) the nested call executes
4) after it ends, the previous exec context is retrieved from the stack and the execution is resumed from where it stopped.

Any recursion can be rewritten in an iterative way ⤳ it may be hard

Recursion: Example

Example : power (2,3)

1 execution, $n \neq 1$ thus we call power(x, n-1) -> power(2,2)

insert context in the stack

Execution stack

-> context {x: 2, n: 1}

-> context {x: 2, n: 2}

-> context {x: 2, n: 3}

2 execution, $n \neq 1$ thus we call power (2,1)

insert context in the stack

3 execution $n == 1$ thus we return x => return 2

↳ remove context from the stack.

Restore the previous call from the top of the stack

-> power(2,2)

- ↳ subcall power(2,1) that already returned 2
- ↳ return $2 * 2 = 4$
- ↳ remove context {x: 2, n: 2} from the stack

Restore the previous call from the stack

context {x: 2, n: 3} ~> power (2,3)

- ↳ subcall power(2,2) that already returned 4
- ↳ $2 * 4 = 8$ -> return 8

The recursion depth in this case was 3.

Mergesort

Merge-sort : An example. Input sequence : $< \overset{\overset{p=1}{\downarrow}}{5}, 2, 4, 7, 1, 3, 2, \overset{\overset{r=8}{\downarrow}}{6} >$

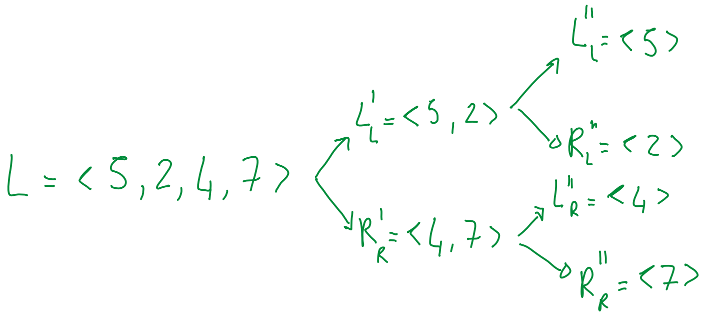
The first operation is a DIVIDE $\rightsquigarrow$ $L = (A, p, q) = (A, 1, q)$ where $q = \lfloor (p+r)/2 \rfloor = 4$

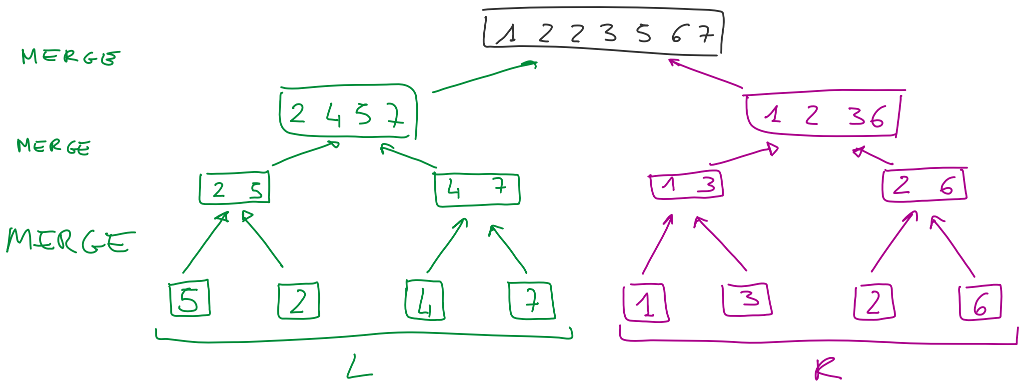
$R = (A, q+1, r) = (A, 1, 5)$

↳ $L = < 5, 2, 4 7 >$ $R = < 1, 3, 2, 6 >$

The 2^{nd} and 3^{rd} operations are RECURSIVE calls of merge-sort

↳ we keep dividing the sequences





```
MERGE-SORT(A, p, r)
  if p < r then
    q = ⌊(p+r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q+1, r)
    MERGE(A, p, q, r)
```

Example

we keep dividing the sequences until $p \geq r$

it's actually q+1 in the second recursive call

when we divide $\boxed{2 \quad 6}$, $r = 8$ and $p = 7 \Rightarrow q = 7$

then we have $\boxed{6}$ with $r = 8$ and $p = 8 \Rightarrow q = 8 \geq r \Rightarrow$ STOP

Merge pseudocode

$p=1$ ↓ $r=8$ ↓

Input sequence : $< 5, 2, 4, 7, 1, 3, 2, 6 >$ $\quad q = \lfloor (p+r)/2 \rfloor$

```
MERGE (A, p, q, r)   ⤳ instance size: $n = r - p + 1$
  $n_1 = q - p + 1$   // set the length of the array
  $n_2 = r - q$       // ↓
  let $L[1, \dots, n_1+1]$ and $R[1, \dots, n_2+1]$ be new arrays
  for $i = 1$ to $n_1$ do              // copy the elements in A in L        $\Theta(n)$
      $L[i] = A[p+i-1]$
  for $j = 1$ to $n_2$ do              // copy the elements in R             $\Theta(n)$
      $R[j] = A[q+j]$
  $L[n_1+1] = \infty$  } sentinels
  $R[n_2+1] = \infty$  }
  $i = 1$
  $j = 1$
  for $K = p$ to $r$ do
      if $L[i] \leq R[j]$ then     } copy the smallest element in A
          $A[K] = L[i]$             }   in this case we take it from L
          $i$++                     } and then we move to the next position
      else
          $A[K] = R[j]$             } in this case we take the element from R
          $j$++                     }
```

This loop iterates over all the elements to be merged ⇓ $\Theta(n)$

The merge procedure takes $\Theta(n)$. Let's analyse MERGESORT.

```
MERGE-SORT(A, p, r)
  if $p < r$ then     ⤳ if $p \geq r$ then A has at most 1 element ⇒ already sorted.
                        else → Keep dividing the sequence until $|A| = 1$ an then MERGE
      $q = \lfloor (p+r)/2 \rfloor$
      MERGE-SORT (A, p, q)
      MERGE-SORT (A, q+1, r)
      MERGE (A, p, q, r)
```

Analysis of the running time

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ a\,T\left(\frac{n}{b}\right) + D(n) + C(n) & \text{otherwise} \end{cases}$$

→ general case for a recursive algorithm

$T(n)$ is the running time of the algorithm. In the case the size of the problem is smaller than a constant c, then the running time is constant ⤳ $\Theta(1)$

Let's suppose that we divide the input instance in a subproblems of size n/b, then each subproblem requires $T(n/b)$ to be solved which is $a\,T(n/b)$ for the whole problem.
To this we need to add the divide time $D(n)$ and the combine time $C(n)$

For analysing MERGE-SORT, let's consider n = power of 2 such that each subproblem is exactly $n/2$ → this assumption has no sizeable impact on the running time

Divide: We split an array in 2, the cost is constant : $D(n) = \Theta(1)$
Conquer: We need to solve $a = 2$ subproblems of size $n/2$, thus $2T(n/2)$
Combine : We saw that the merge procedure takes $\Theta(n)$

$$\downarrow \quad T(n) = \begin{cases} \Theta(1) \text{ if } n \leq c \Rightarrow n = 1 \\ 2\,T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$