

Recursion uses the algorithmic strategy **divide-and-conquer**. In general, this strategy looks like this:

```
1 solve p =
2   divide p in parts p1, p2, ..., pn
3   solve each part, resulting in the partial solutions s1, s2, ..., sn
4   do some preprocessing in each partial solution
5   join the the partial solutions to make the solution for p
```

Usually, solving each part is just making a recursive call. Then the hard work is finding out how to join the partial results. Let's apply this strategy for problems with lists. First, how can we divide a list? In the middle, for example, resulting in two smaller lists. But the simplest way is to divide into head (the first element) and tail (the list with the remaining elements). Notice that the empty list cannot be divided. For example, how can we define a function `sum p` to sum up the elements of a list `p`?

First, we devide `p` in parts `p1` (head) and `p2` (tail) - we do this in Haskell by `p1=head p` and `p2=tail p` in a `let` environment. Now we solve the parts: as `p1` is only one number, it is "already solved", and so `s1=p1`; `p2` is a list, and we call the function we are defining to solve it: `s2=sum p2`. Now we need to join the partial solutions `s1` and `s2`. We note that `s1` is just the first element and `s2` is the sum of the tail of the original list `p`. So with little thinking we conclude that the joining is done by just adding `s1` and `s2`.

```
1 sumAll p =
2   let p1=head p; p2= tail p
3       s1=p1; s2= sumAll p2
4       --there's no need for processing the partial solutions
5   in s1+s2
```

What happen if the function is called with the empty list? As it cannot be divided, it will result in a runtime error. It might be convenient to treat it as a special case

and just return zero. Note also that if the list has only one element, there's nothing to add and we treat it as a special case, just returning its element. These special (or trivial) cases are used to stop the recursion - that is, their solution is so easy that there is no need to divide it further. Including these cases, we have:

```
1 sumAll [] = 0
2 sumAll [x] = x
3 sumAll p =
4   let p1=head p; p2= tail p
5       s1=p1; s2= sumAll p2
6   in s1+s2
```

or, without the let construction:

```
1 sumAll [] = 0
2 sumAll (p1:p2) = p1+sumAll p2
```

Sometimes we need a little more thinking to find the right joining function. For example, suppose we want to use this strategy to solve the string2words problem: we divide a string *s* into *c* (head) and *t* (tail). If we have the solution for *t*, how do we join with *c*? For example, if *s* is "one two three", the *c* is 'o' and the partial solution for *t* is ["ne", "two", "three"]. Then joining the partial results seems just including *c* in the first element of *c*. Try this solution and see if it works.

In the following exercises, you should use the divide-and-conquer strategy wherever possible.

1) For each of the following summations, make a function that receive as argument an integer number n and returns the value of the summation:

$$\text{a) } \sum_{i=1}^n (2i - 1) \quad \text{b) } \sum_{i=0}^n 2i \quad \text{c) } \sum_{i=0}^n i^2 \quad \text{d) } \sum_{i=0}^n i^3$$

$$\text{e) } \sum_{i=1}^n 2^i \quad \text{f) } \sum_{i=1}^n \frac{1}{\sqrt{i}} \quad \text{g) } \sum_{i=1}^n \frac{1}{i} \quad \text{h) } \sum_{i=1}^n \frac{1}{i^i}$$

$$\text{i) } \sum_{i=1}^n \frac{1}{i(i+1)} \quad \text{j) } \sum_{i=1}^n \frac{1}{(2i+1)(2i-1)} \quad \text{k) } \sum_{i=1}^n \frac{1}{i(i+3)}$$

$$\text{l) } \sum_{i=0}^n \frac{2^i + 3^i}{6^i} \quad \text{m) } \sum_{i=1}^n \frac{1}{i(i+1)(i+2)}$$

2) The following transcendental functions can be approximate by series. For each one, define a function that takes two arguments x (double) and n (integer) and returns the value of the series after n iterations:

a) $\text{sine}(x)$ - sine of the angle x

b) $\text{cosine}(x)$ - cosine of the angle x

c) $\text{expt}(x)$ - exponentiation: e^x

d) $\text{logn}(x)$ - natural logarithm: $\ln(x)$

The infinite series for these functions are:

$$\text{a) } \text{sen}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$\text{b) } \text{cos}(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

$$\text{c) } \text{exp}(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

$$\text{d) } \ln(x) = 2 \left[\left(\frac{x-1}{x+1} \right) + \frac{1}{3} \left(\frac{x-1}{x+1} \right)^3 + \frac{1}{5} \left(\frac{x-1}{x+1} \right)^5 + \dots \right]$$

3) Define a function that takes an integer argument and returns a integer that corresponds to the reverse of the number. For example, $\text{rev}(134)$ returns 431 and $\text{rev}(130)$

returns 31.

4) Define functions that take as argument a list x and returns:

- a) the number of elements in x
- b) the greatest element of x
- c) the last element of x
- d) true or false, depending on x being ordered or not
- e) another list with the elements of x without repetition
- f) the mean of the values in x
- g) another list with the elements of x added two by two:
the first element added with the second, the second with the third, ...

5) Define functions that take 2 arguments: a list y and an element x . The function should return:

- a) true, if x belongs to the list, and false otherwise
- b) the number of times that x occurs in the list
- c) the position of the first occurrence of x in the list
- d) a new list with x inserted at the end of y
- e) a new list with the elements of y incremented by x
- f) the element of the x^{th} position of y

6) Define a function that takes 3 arguments: a list, an element x and an integer n . The function inserts x at the position number n of the list.

7) Define functions that take 2 arguments: a sorted list y and an element x . The function inserts x in the right position of y , so that y is still ordered.

8) Now you define functions that take as arguments two lists x and y . Each function returns:

- a) true if all the elements of x are also elements of y , and false otherwise
- b) true if x is a prefix of y , and false otherwise (this means to verify if there exists z such that $x++z=y$)
- c) true if x is a postfix of y , and false otherwise (this means to verify if there exists z such that $z++x=y$)
- d) true if x is a "sublist" of y , and false otherwise (this means to verify if there exists z and w such that $w++x++z=y$)
- e) a list with the elements of x and y added pairwise - that is the first element of x added with the first element of y , the second element of x added with the second element of y ,...

9) In a supermarket, goods and their prices can be represented as a list of pairs, where the first component of the pair is the name of the article, and the second is the price. For example, `[("beans",145),("wheat",32),("passion fruit",63)]`. Let `lp` be a list of this type. Define the following functions:

- a) take `lp` and an article `p` as arguments and return the price of `p`
- b) take `lp`, an article `p` and a quantity `n` as arguments and returns the price of `n` units of the article `p`
- c) take `lp` and a list with a list of goods to buy and returns the total cost of the shopping
- d) take `lp`, an article `p` and a new price `x` as arguments and update the the price of `p` in `lp`

10) Define functions that take an integer `n` as argument and returns a string with

- a) base 10 (decimal) representation of `n`
- b) base 16 (hexadecimal) representation of `n`
- c) base 2 (binary) representation of `n`

11) Generalize the former problem: define a function that takes 2 arguments: two integers `n` and `k`. The function returns the string with the representation of `n` in the base `k`. Use letters for bases grater than 10, starting in 'A' for 10.

12) Define the inverse of the former function: the inputs are a string `s` and the base `k` and the output is the corresponding integer.

13) For the following items, define functions that take as arguments a string `s` and a character `c` and returns:

- a) the string `s` without any occurrency of `c`
- b) the number of times that `c` occurs in `s`
- c) the character that occurs immediatly after `c` in `s`. If `c` is the last character (of if `c` does not occur in `s`), the null character `'\0'` is returned