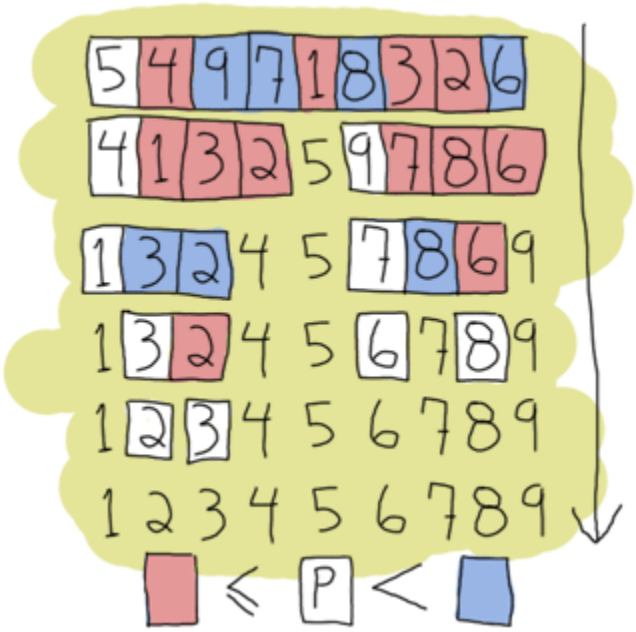
QUICK SORT IN ERLANG



I can (and will) now assume recursion and tail recursion make sense to you, but just to make sure, I'm going to push for a more complex example, quicksort. Yes, the traditional "hey look I can write short functional code" canonical example. A naive implementation of quicksort works by taking the first element of a list, the *pivot*, and then putting all the elements smaller or equal to the pivot in a new list, and all those larger in another list. We then take each of these lists and do the same thing on them until each list gets smaller and smaller. This goes on until you have nothing but an empty list to sort, which will be our base case. This implementation is said to be naive because smarter versions of quicksort will try to pick optimal pivots to be faster. We don't really care about that for our example though.
We will need two functions for this one: a first function to partition the list into smaller and larger parts and a second function to apply the partition function on each of the new lists and to glue them together. First of all, we'll write the glue function:

```
quicksort([]) -> [];
quicksort([Pivot|Rest]) ->
{Smaller, Larger} = partition(Pivot,Rest,[],[]),
quicksort(Smaller) ++ [Pivot] ++ quicksort(Larger
).
```

This shows the base case, a list already partitioned in larger and smaller parts by another function, the use of a pivot with both lists quicksorted appended before and after it. So this should take care of assembling lists. Now the partitioning function:

```
partition(_,[],   Smaller,   Larger) -> {Smaller, Larger};
partition(Pivot, [H|T], Smaller, Larger) ->
if H =< Pivot -> partition(Pivot,  T,  [H|Smaller], Larger);
H > Pivot -> partition(Pivot,     T,     Smaller, [H|Larger])
end.
```

And you can now run your quicksort function. If you've looked for Erlang examples on the Internet before, you might have seen another implementation of quicksort, one that is simpler and easier to read, but makes use of list comprehensions. The easy to replace parts are the ones that create new lists, the `partition/4` function:

```
lc_quicksort([]) -> [];
lc_quicksort([Pivot|Rest]) ->
lc_quicksort([Smaller || Smaller <- Rest, Smaller =< Pivot])
++ [Pivot] ++
lc_quicksort([Larger || Larger <- Rest, Larger > Pivot]).
```

The main differences are that this version is much easier to read, but in exchange, it has to traverse the list twice to partition it in two parts. This is a fight of clarity against performance, but the real loser here is you, because a function `lists:sort/1` already exists. Use that one instead.

Don't drink too much Kool-Aid:

All this conciseness is good for educational purposes, but not for performance. Many functional programming tutorials never mention this! First of all, both implementations here need to process values that are equal to the pivot more than once. We could have decided to instead return 3 lists: elements smaller, larger and equal to the pivot in order to make this more efficient.

Another problem relates to how we need to traverse all the partitioned lists more than once when attaching them to the pivot. It is possible to reduce the overhead a little by doing the concatenation while partitioning the lists in three parts. If you're curious about this, look at the last function (`bestest_qsort/1`) of recursive.erl for an example.

A nice point about all of these quicksorts is that they will work on lists of any data type you've got, even tuples of lists and whatnot. Try them, they work!

Source : http://learnyousomeerlang.com/recursion