Here are some extra programming problems that can be done using the material in this module. Many are similar in difficulty and content to the homework, but they are not the homework, so you are free to discuss solutions, etc. on the discussion forum. Thanks to Charilaos Skiadas for contributing these.

1. Write a function `alternate : int list -> int` that takes a list of numbers and adds them with alternating sign. For example `alternate [1,2,3,4] = 1 - 2 + 3 - 4 = -2`.
2. Write a function `min_max : int list -> int * int` that takes a non-empty list of numbers, and returns a pair `(min, max)` of the minimum and maximum of the numbers in the list.
3. Write a function `cumsum : int list -> int list` that takes a list of numbers and returns a list of the partial sums of those numbers. For example `cumsum [1,4,20] = [1,5,25]`.
4. Write a function `greeting : string option -> string` that given a string option `SOME` *name* returns the string `"Hello there, ...!"` where the dots would be replaced by *name*. Note that the name is given as an option, so if it is `NONE` then replace the dots with `"you"`.
5. Write a function `repeat : int list * int list -> int list` that given a list of integers and another list of nonnegative integers, repeats the integers in the first list according to the numbers indicated by the second list. For example: `repeat ([1,2,3], [4,0,3]) = [1,1,1,1,3,3,3]`.
6. Write a function `addOpt : int option * int option -> int option` that given two "optional" integers, adds them if they are both present (returning `SOME` of their sum), or returns `NONE` if at least one of the two arguments is `NONE`.
7. Write a function `addAllOpt : int option list -> int option` that given a list of "optional" integers, adds those integers that are there (i.e. adds all the `SOME i`). For example: `addOpt ([SOME 1, NONE, SOME 3]) = SOME 4`. If the list does not contain any `SOME` is in it, i.e. they are all `NONE` or the list is empty, the function should return `NONE`.
8. Write a function `any : bool list -> bool` that given a list of booleans returns `true` if there is at least one of them that is `true`, otherwise returns `false`. (If the list is empty it should return `false` because there is no `true`.)
9. Write a function `all : bool list -> bool` that given a list of booleans returns `true` if all of them `true`, otherwise returns `false`. (If the list is empty it should return `true` because there is no `false`.)
10. Write a function `zip : int list * int list -> int * int` list that given two lists of integers creates consecutive pairs, and stops when one of the lists is

empty. For example: `zip ([1,2,3], [4, 6]) = [(1,4), (2,6)]`.

11. Challenge: Write a version `zipRecycle` of `zip`, where when one list is empty it starts recycling from its start until the other list completes. For example: `zipRecycle ([1,2,3], [1, 2, 3, 4, 5, 6, 7]) = [(1,1), (2,2), (3, 3), (1,4), (2,5), (3,6), (1,7)]`.
12. Lesser challenge: Write a version `zipOpt` of `zip` with return type `(int * int) list option`. This version should return `SOME` of a list when the original lists have the same length, and `NONE` if they do not.
13. Write a function `lookup : (string * int) list * string -> int option` that takes a list of pairs `(s, i)` and also a string `s2` to look up. It then goes through the list of pairs looking for the string `s2` in the first component. If it finds a match with corresponding number `i`, then it returns `SOME i`. If it does not, it returns `NONE`.
14. Write a function `splitup : int list -> int list * int list` that given a list of integers creates two lists of integers, one containing the non-negative entries, the other containing the negative entries. Relative order must be preserved: All non-negative entries must appear in the same order in which they were on the original list, and similarly for the negative entries.
15. Write a version `splitAt : int list * int -> int list * int list` of the previous function that takes an extra "threshold" parameter, and uses that instead of 0 as the separating point for the two resulting lists.
16. Write a function `isSorted : int list -> boolean` that given a list of integers determines whether the list is sorted in increasing order.
17. Write a function `isAnySorted : int list -> boolean`, that given a list of integers determines whether the list is sorted in either increasing or decreasing order.
18. Write a function `sortedMerge : int list * int list -> int list` that takes two lists of integers that are each sorted from smallest to largest, and merges them into one sorted list. For example: `sortedMerge ([1,4,7], [5,8,9]) = [1,4,5,7,8,9]`.
19. Write a sorting function `qsort : int list -> int list` that works as follows: Takes the first element out, and uses it as the "threshold" for `splitAt`. It then recursively sorts the two lists produced by `splitAt`. Finally it brings the two lists together. (Don't forget that element you took out, it needs to get back in at some point). You could use `sortedMerge` for the "bring together" part, but you do not need to as all the numbers in one list are less than all the numbers in the other.)
20. Write a function `divide : int list -> int list * int list` that takes a list of integers and produces two lists by alternating elements between the two lists. For example: `divide ([1,2,3,4,5,6,7]) = ([1,3,5,7], [2,4,6])`.
21. Write another sorting function `not_so_quick_sort : int list -> int list` that works as follows: Given the initial list of integers, splits it in two lists using

divide, then recursively sorts those two lists, then merges them together with `sortedMerge`.

22. Write a function `fullDivide` : `int` * `int` -> `int` * `int` that given two numbers `k` and `n` it attempts to evenly divide `k` into `n` as many times as possible, and returns a pair (`d`, `n2`) where `d` is the number of times while `n2` is the resulting `n` after all those divisions. Examples: `fullDivide` (2, 40) = (3, 5) because 2*2*2*5 = 40 and `fullDivide`((3,10)) = (0, 10) because 3 does not divide 10.
23. Using `fullDivide`, write a function `factorize` : `int` -> (`int` * `int`) `list` that given a number `n` returns a list of pairs (`d`, `k`) where `d` is a prime number dividing `n` and `k` is the number of times it fits. The pairs should be in increasing order of prime factor, and the process should stop when the divisor considered surpasses the square root of `n`. If you make sure to use the reduced number `n2` given by `fullDivide` for each next step, you should not need to test if the divisors are prime: If a number divides into `n`, it must be prime (if it had prime factors, they would have been earlier prime factors of `n` and thus reduced earlier). Examples: `factorize`(20) = [(2,2), (5,1)]; `factorize`(36) = [(2,2), (3,2)]; `factorize`(1) = [].
24. Write a function `multiply` : (`int` * `int`) `list` -> `int` that given a factorization of a number `n` as described in the previous problem computes back the number `n`. So this should do the opposite of `factorize`.
25. Challenge (hard): Write a function `all_products` : (`int` * `int`) `list` -> `int list` that given a factorization list result from `factorize` creates a list all of possible products produced from using some or all of those prime factors no more than the number of times they are available. This should end up being a list of all the divisors of the number `n` that gave rise to the list. Example: `all_products`([(2,2), (5,1)]) = [1,2,4,5,10,20]. For extra challenge, your recursive process should return the numbers in this order, as opposed to sorting them afterwards.

Mark as completed