```
(*
                              CS51 Lab 10
           Time Complexity, Big-O, and Recurrence Equations
 *)

(* Objective:

This lab is intended to practice concepts concerning efficiency and
complexity, including:
    Big O notation
    Recurrence equations
 *)

(* We open `CS51Utils` to provide access to the `Absbook` module, and
   timing functions such as `Absbook.call_timed`. You installed the
   `CS51Utils` package at the beginning of the course. You'll find the
   `Absbook` module source code in the file `absbook.ml` in the CS51
   utilities repository at <https://github.com/cs51/utils>. *)

open CS51Utils ;;

(*======================================================================
Part 1: Empirical analysis of functions

In the reading, we empirically determined the efficiency of mergesort
and insertion sort by timing these functions on the same inputs of
various lengths. The ability to perform empirical analysis of programs
will often prove useful.

    Throughout this lab you may find various functions in the
    `Absbook` module to be helpful, as well as OCaml's `Random`
    library module
    <https://caml.inria.fr/pub/docs/manual-ocaml/libref/Random.html>.

    In order to make use of the `Absbook` module from within `ocaml`
    or `utop`, you may need to inform these REPLs of information about
    the module's location. If you have trouble accessing `Absbook`
    functions, try the commands in the following example:

        % ocaml
                OCaml version 4.11.1

        # #use "topfind" ;;
        ...
        - : unit = ()
        # #require "CS51Utils" ;;
        ...
        # open CS51Utils ;;
        # Absbook.call_timed ;;
        - : ?count:int -> ('a -> 'b) -> 'a -> 'b * float = <fun>

......................................................................
Exercise 1: Write a function `random_list` that creates a list of a
specified length of random integers between 0 and 999 inclusive. Hint:
You may find the `List.init` function to be helpful.
......................................................................*)

let random_list (length : int) : int list =
  failwith "random_list not implemented" ;;

(*......................................................................
Exercise 2: Write a function `time_sort` that, given an `int list ->
int list` sorting function and a list of integers, returns a `float`
indicating how long in milliseconds that the sort takes. Hint: You may
find the `Absbook.call_timed` function to be helpful.
```

```
.............................................................*)

let time_sort (sort : int list -> int list) (lst : int list) : float =
  failwith "time_sort not implemented";;

(* We've provided implementations of merge sort and insertion sort
here as modules satisfying the `SORT` signature so that you have some
things to time. *)

module type SORT =
  sig
    (* sort xs -- Return the list `xs` sorted in increasing
       order by OCaml's `<` operator. *)
    val sort : 'a list -> 'a list
  end ;;

module InsertSort : SORT =
  struct
    let rec insert (xs : 'a list) (x : 'a) : 'a list =
      match xs with
      | [] -> [x]
      | hd :: tl -> if x < hd then x :: xs
                    else hd :: (insert tl x) ;;

    let rec sort (xs : 'a list) : 'a list =
      match xs with
      | [] -> []
      | hd :: tl -> insert (sort tl) hd ;;
  end ;;

module MergeSort : SORT =
  struct
    let rec split (xs : 'a list) : 'a list * 'a list =
      match xs with
      | [] -> [], []
      | [x] -> [x], []
      | first :: second :: rest -> let rest1, rest2 = split rest in
                                   first :: rest1, second :: rest2

    let rec merge (xs : 'a list) (ys : 'a list) : 'a list =
      match xs, ys with
      | [], _ -> ys
      | _, [] -> xs
      | x :: xs_tl, y :: ys_tl ->
         if x < y then x :: (merge xs_tl ys)
         else y :: (merge xs ys_tl)

    let rec sort (xs : 'a list) : 'a list =
      match xs with
      | []
      | _ :: [] -> xs
      | _ -> let first, second = split xs in
             merge (sort first) (sort second)
  end ;;

(*.............................................................
Exercise 3: How many functions does the `InsertionSort` module
provide? How many functions does the `MergeSort` module
provide? Define the variables below accordingly (replacing the `-1`
values).
.............................................................*)

let insertion_sort_provides : int = -1 ;;
let merge_sort_provides : int = -1 ;;
```

```
(*.....................................................................
Exercise 4: Compare the time it takes for merge sort and insertion
sort to run on lists of random ints of length 10 and 1000. Use the
implementation of merge and insertion sort above.
.....................................................................*)

(* Fill in the table below:

                |    List length 10    |  List length 1000
                |    Time (msecs)      |  Time (msecs)
-------------------------------------------------------------
Insertion Sort  |                      |
-------------------------------------------------------------
Merge Sort      |                      |
-------------------------------------------------------------
 *)

(*=====================================================================
Part 2: Big-O

.....................................................................
Exercise 5: In the reading for this lab, we saw that big-O notation is
a generic way of expressing the growth rate of a function. For each of
the functions defined below, state in which big-O class(es) the
function belongs.

To allow us to check answers, we've defined an OCaml data type,
`complexity`, with various commonly used big-O classes. The
name-to-function mapping is set out below. We will use informal
function notation in this lab. For more on informal versus formal
notation, please refer to the reading for today's lab.

    Constant    -> O(1)
    Logarithmic -> O(log(n))
    Linear      -> O(n)
    LogLinear   -> O(n log(n))
    Quadratic   -> O(n^2)
    Cubic       -> O(n^3)
    Exponential -> O(2^n)

Because functions can be in more than one complexity class, the format
of the solution to each exercise is a list of complexity classes. By
way of example, we've done the first problem (Exercise 5a) for you.
.....................................................................*)

type complexity =
    | Constant
    | Logarithmic
    | Linear
    | LogLinear
    | Quadratic
    | Cubic
    | Exponential ;;

(* f(x) = 5^x + x^3 *)
let exercise5a () : complexity list =
  [Exponential] ;;

(* f(x) = 0 *)
let exercise5b () : complexity list =
  failwith "exercise5b not implemented" ;;

(* f(x) = 3 x^2 + 2 x + 4 *)
let exercise5c () : complexity list=
  failwith "exercise5c not implemented" ;;
```

```
(* f(x) = (2 x - 3) log(x) + 100 x *)
let exercise5d () : complexity list =
  failwith "exercise5d not implemented" ;;

(* f(x) = x (x^2 + x) *)
let exercise5e () : complexity list =
  failwith "exercise5e not implemented" ;;

(* One advantage of big-O is that we can disregard constants in
considering asymptotic performance of functions. We saw empirically
that on large inputs, merge sort worked faster than insertion
sort. The ability to disregard constants tells us that merge sort will
eventually be faster than insertion sort, even if we add a constant
amount of time to merge sort's performance.

Let's actually do this and test the results empirically!

Here is a version of merge sort that inserts a small constant delay
(50 milliseconds), to simulate a version of the function with the same
asymptotic complexity but that is a constant amount slower. *)

module DelayMergeSort : SORT =
  struct
    (* DelayMergeSort first sleeps for a predetermined period of time,
       then runs our generic MergeSort sort.  This sleep will add a
       constant amount of time to each run of DelayMergeSort.  *)
    let sort (xs : 'a list) : 'a list =
      let () = Unix.sleepf 0.05 in
      MergeSort.sort xs  ;;
  end ;;

(*.....................................................................
Exercise 6: Additive constants

Fill in the table below.
.....................................................................*)

(*                |    List length 10    |  List length 1000
                  |    Time (msecs)      |  Time (msecs)
-------------------------------------------------------------
Insertion Sort    |                      |
-------------------------------------------------------------
Delay Merge Sort  |                      |
-------------------------------------------------------------*)

(* You likely found that InsertSort was faster than DelayMergeSort,
even on a list of length 1000. Increase the length of the list being
sorted by DelayMergeSort and InsertSort until DelayMergeSort runs
faster than InsertSort. Record the size of a list for which this is
true below. *)

let exercise6 = 0    (* replace with the requested list size *)

(* Big-O also allows us to disregard constant *multiplicative*
factors. In the next exercise, we work with a version of MergeSort
that sorts a given list twice rather than once.  However long
MergeSort takes, DoubleMergeSort will thus take twice as long. *)

module DoubleMergeSort : SORT =
  struct
    (* By sorting the list twice, we double the time MergeSort
       takes *)
    let sort (xs : 'a list) : 'a list =
      let _ = MergeSort.sort xs in
```

```
      MergeSort.sort xs  ;;
  end ;;

(*.....................................................................
Exercise 7: Multiplicative constant factors

Complete the same empirical analysis as above to compare the
asymptotic behavior of InsertSort and DoubleMergeSort, and fill in the
table below.
.....................................................................*)

(*                 |    List length 10    |  List length 1000
                   |    Time (msecs)      |  Time (msecs)
-------------------------------------------------------------
Insertion Sort     |                      |
-------------------------------------------------------------
Double Merge Sort  |                      |
-------------------------------------------------------------*)

(* Now record a list length for which you found DoubleMergeSort sorted
faster than InsertSort. *)

let exercise7 = 0    (* replace with the requested list size *)

(* An additional nice property of big-O is the ability to disregard
lower-order terms of a function. In the reading, we found that:

    Time_mergesort(n) = c * n log n + d

In this exercise, we will work with a version of MergeSort that will
add an additional k * n term to the completion time of MergeSort. *)

module ExtraTermMergeSort : SORT =
  struct
    let sort (xs : 'a list) : 'a list =
      (* We map the identity function over all of the elements of the
         list and throw away the result, so as to waste time of O(n),
         where n is the number of elements in the list *)
      let _ = List.map (fun x -> x) xs in
      MergeSort.sort xs;;
  end ;;

(*.....................................................................
Exercise 8: Lower Order Terms

Complete the same empirical analysis as above to compare the asymptotic
behavior of InsertSort and ExtraTermMergeSort, and fill in the table
below.
.....................................................................*)

(*                    |    List length 10    |  List length 1000
                      |    Time (msecs)      |  Time (msecs)
----------------------------------------------------------------
Insertion Sort        |                      |
----------------------------------------------------------------
Extra Term Merge Sort |                      |
----------------------------------------------------------------*)

(* Now record a list length for which ExtraTermMergeSort works faster
than InsertSort. *)
let exercise8 = 0    (* replace with the requested list size *)

(*.....................................................................
Exercise 9: More big-O
```

As in Exercise 4, for each of the functions below, state to which
big-O classes the function belongs. See Exercise 4 for an example.

As a reminder, the big-O classes defined in our complexity ADT are

```
    type complexity =
         | Constant
         | Logarithmic
         | Linear
         | LogLinear
         | Quadratic
         | Cubic
         | Exponential ;;
.....................................................................*)

(* f(x) = 10000 *)
let exercise9a () : complexity list =
  failwith "exercise9a not implemented" ;;

(* f(x) = 50x^100 + x^2 *)
let exercise9b () : complexity list =
  failwith "exercise9b not implemented" ;;

(* f(x) = 30xlog(x) + 50x + 70 *)

let exercise9c () : complexity list =
  failwith "exercise9c not implemented" ;;

(* f(x) = 30x^2 * log(x) *)
let exercise9d () : complexity list =
  failwith "exercise9d not implemented" ;;

(* f(x) = x + 60log(x) *)
let exercise9e () : complexity list =
  failwith "exercise9e not implemented" ;;

(*=====================================================================
```

Part 3: Recurrence Equations

Once we know the complexity of a function, we can use big-O notation
to compare that function’s asymptotic performance with other
functions. However, a function’s complexity may not be immediately
obvious. Recurrence equations provide an analytical way to determine
the complexity of a function.

Recurrence equations generally consider two cases:

1. A base case
2. A recursive case

Once you have formulated the recursive case, you can use the method of
"unfolding" described in the reading to determine the time complexity
of the functions.

In each of the exercises below, we present a function in OCaml. Your
task is to define the recurrence equations for that function, and then
to solve the recurrence equations via unfolding, to generate a
closed-form equation and form a conclusion about the time complexity
of the function in big-O notation.

To facilitate automated testing of the recurrence equations you come
up with, we ask you to present them in the form of an OCaml
function. (We provide an example below.) We encourage you to first
complete the problem on paper, with the notation from the chapter and
using the unfolding method, and then transfer your solutions

```
here. When finding the time complexity, we would like you to use the
tightest possible big-O class.

Many of the recurrences have various constants. We have defined a
global variable, `k`, for you to use for *all* of the constants in your
OCaml formulation of the recurrence equations. Again, the example
below should clarify. *)

let k = 5;;

(* An example of the method we would like you to use for presenting
your recurrence equations and complexity is provided below, based on
the `insert` function from the reading. *)

(*.....................................................................
                        RECURRENCE EXAMPLE
.....................................................................*)
let rec insert xs x =
    match xs with
    | [] -> [x]
    | h :: t -> if x > h then h :: (insert t x)
                  else x :: xs ;;
(*.....................................................................
Complete the recurrence equations and time complexity of this
function:
.....................................................................*)

(*
let time_insert (n : int) : int =
    failwith "time_insert not yet implemented" ;;

let insert_complexity () : complexity =
    failwith "insert_complexity not yet implemented" ;;
*)

(*.....................................................................
                            SOLUTION

We saw in the reading that the `insert` function has the following
recurrence equations:

T_insert(0) = c
T_insert(n) = max(k_1 + T_insert(n-1), k_2) â\211¤ k_1 + T_insert(n-1) + k_2
            = k + T_insert(n-1)

We express these equations with the following Ocaml function. Rather
than write two equations, as you might on paper, we write the base
case and recursive case as branches of an if/then/else
expression. Although the recurrence equations have two constants (c
and k), we use the single OCaml variable `k` to play both of those
roles.  Please follow that format in all of the following exercises.
.....................................................................*)

let rec time_insert (n : int) : int =
  if n = 0 then k
  else k + time_insert (n - 1) ;;

(* (Note that this function will run forever on inputs below 0. We
would normally expect you to handle invalid inputs. However, for the
purpose of practicing recurrence equations, you may assume inputs will
always be positive or nonnegative as the equations require.)

For the time complexity, we use a value from the complexity type to
express the tightest big-O bound for this linear function. *)
```

```
let insert_complexity () : complexity =
  Linear ;;

(* Now it's time for you to construct and solve some recurrence
equations, and add them to the lab using the method above.

                            END OF EXAMPLE
 *)

(*.....................................................................
Exercise 10: Sum recurrence equations

Formulate the recurrence equations and determine the time complexity
of the `sum` function, defined below.
.....................................................................*)

let rec sum (x : int list) : int =
  match x with
  | [] -> 0
  | h :: t -> h + sum t;;

(* Describe the time complexity recurrence equations for `sum` as an
OCaml function of `n`, the length of the list. *)
let time_sum (n : int) : int =
  failwith "time_sum not yet implemented" ;;

(* What is its complexity? *)
let sum_complexity () : complexity =
  failwith "sum_complexity not yet implemented" ;;

(*.....................................................................
Exercise 11: Divider Recurrence Equations

Formulate the recurrence equations and determine the time complexity
of `divider`, defined below, in terms of the value of its argument
`x`.
.....................................................................*)

let rec divider (x : int) : int =
  if x < 0 then
    raise (Invalid_argument "only positive numbers accepted")
  else if x <= 1 then 0
  else 1 + divider (x / 2) ;;

let time_divider (n : int) : int =
  failwith "time_sum not yet implemented" ;;

let divider_complexity () : complexity =
  failwith "time_complexity not yet implemented" ;;

(*.....................................................................
Exercise 12: Find_min recurrence equations

Formulate the recurrence equations and determine the time complexity
of `find_min` as defined below. The `find_min` function contains a
helper function, `split`. First find the recurrence equations and time
complexity of `split`, as you may find this helpful in determining the
time complexity of `find_min`.
.....................................................................*)

let rec find_min (xs : int list) : int =

  let rec split (xs : 'a list) =
    match xs with
    | [] -> [], []
```

```
    | [x] -> [x], []
    | first :: second :: rest -> let rest1, rest2 = split rest in
                                 first :: rest1, second :: rest2 in

  match xs with
  | [] -> raise (Invalid_argument "Empty List")
  | [x] -> x
  | _ -> let xs1, xs2 = split xs in
         min (find_min xs1) (find_min xs2) ;;

let time_split (n : int) : int =
  failwith "time_split not yet implemented" ;;

let split_complexity () : complexity =
  failwith "split_complexity not yet implemented" ;;

let time_find_min (n : int) : int =
  failwith "time_find_min not yet implemented" ;;

let find_min_complexity () : complexity =
  failwith "find_min_complexity not yet implemented" ;;

(*======================================================================
Part 4: Tradeoffs

Consider the three implementations of multiplication below:
 *)

let sign_product (x : int) (y : int) : int =
    let sign (x : int) : int =
        if x >= 0 then 1 else ~-1 in
    (sign x) * (sign y) ;;

(* 1. Repeated addition simply adds `x` to itself `y` times. To
multiply two n-digit numbers together using this algorithm takes time
O(n 10^n). It is thus tremendously inefficient. *)

let mult_repeated_addition (x : int) (y : int) : int =
  let rec helper (y : int) (sum : int) =
    if y < 0 then raise (Invalid_argument "negative input")
    else if y = 0 then 0
    else helper (y - 1) (sum + (abs x)) in
  helper (abs y) 0 * (sign_product x y) ;;

(* 2. The gradeschool multiplication algorithm is the algorithm one
would use when doing multiplication of large numbers on paper. To
multiply two n-digit numbers together using this algorithm takes time
O(n^2) *)

let mult_grade_school (x : int) (y : int) : int =
  let product_sign = sign_product x y in
  let x = abs x in
  let y = abs y in
  let base = 10 in
  let single_digit_mult (x : int) (y : int) =
    let rec helper x carry placevalue =
      if x = 0 then carry * placevalue
      else let prod = y * (x mod base) + carry in
           let new_carry, place = (prod / base, prod mod base) in
           place * placevalue +
             helper  (x / base) new_carry (placevalue * base)
    in if y < 0 || y > 9 then
         raise (Invalid_argument "multiple digit or neg. y")
       else helper x 0 1 in
  let rec iterate_y y placevalue =
```

```
    if y = 0 then 0
    else placevalue * single_digit_mult x (y mod 10)
         + iterate_y (y / 10) (placevalue * base) in
  product_sign * iterate_y y 1 ;;

(* 3. The Karatsuba algorithm is a multiplication algorithm that
utilizes a divide-and-conquer approach. It was divised in 1960 by
23-year-old student Anatoly Karatsuba after his Professor Andrey
Kolmogorov conjectured that the fastest possible multiplication
algorithm would be lower bounded (that is, no faster than) O(n^2).
Karatsuba's algorithm disproved that conjecture. To multiply two
n-digit numbers, Karatsuba's algorithm runs in O(n^(log_2 3)), that
is, n to the power of log base 2 of 3, which is about n^1.4.

*You do not need to understand the algorithm for this class*, but can
find out more on Wikipedia if interested:
https://en.wikipedia.org/wiki/Karatsuba_algorithm *)

let mult_karatsuba (x : int) (y : int) : int =
  let tens_power (power : int) : int =
    let num_string = "1" ^ (String.make power '0') in
    int_of_string num_string in
  let rec karatsuba x y =
    if x < 10 || y < 10 then  x * y
    else let num_digits_x = String.length (string_of_int x) in
         let num_digits_y = String.length (string_of_int y)  in
         let min_digits = min num_digits_x num_digits_y in
         let half_place = tens_power (min_digits / 2) in
         let highx, lowx = (x / half_place, x mod half_place) in
         let highy, lowy = (y / half_place, y mod half_place) in
         let a = karatsuba lowx lowy in
         let b = karatsuba (lowx + highx) (lowy + highy) in
         let c = karatsuba highx highy in
         c * half_place * half_place
         + half_place * (b - c - a) + a in
  (sign_product x y) * karatsuba (abs x) (abs y) ;;

(*.....................................................................
Exercise 13: Write a function `time_multiply` that, given a
multiplication function and two integers, times how long in seconds
the algorithm takes to multiply the integers.
.....................................................................*)

let time_multiply (mult : int -> int -> int)
                  (x : int)
                  (y : int)
                : float =
  failwith "time_multiply not yet implemented";;

(*.....................................................................
Exercise 14: Fill in the table below:
.....................................................................*)

(*                      |    15 * 50           |  1241342345 *
                        |                      |  3237461243
                        |    Time (msecs)      |  Time (msecs)
-------------------------------------------------------------------
Repeated Addition       |                      |
-------------------------------------------------------------------
Grade School Algorithm  |                      |
-------------------------------------------------------------------
Karatsuba               |                      |
-------------------------------------------------------------------
OCaml Native ( * )      |                      |
-------------------------------------------------------------------
```

```
 *)

(* Questions to consider:

      1. Which algorithm above was easiest to understand?
      2. Which algorithm was likely easiest to code?
      3. Which was fastest on small numbers?
      4. Which was fastest on large numbers?
      5. What size integers do you typically multiply?
      6. Which algorithm, then, would you consider the best?
 *)
```