1. Floating Point

1. Watch the following video: Fast Inverse Square Root – A Quake III Algorithm `https://youtu.be/p8u_k2LIZyo`.

   Where does the constant `0x5f3759df` come from?

   (Just a quick answer, otherwise watching the video isn't much of a question. But watching the video is the main bit of work, not writing down this. You can watch the video on 2× speed if you can still follow it. And you don't have to know the fast inverse square root for the exam.)

   [medium, because of video length]

First note the positive floating point value $x = \left(1 + \frac{m}{2^{23}}\right) \times 2^{e-127}$ has an IEEE 754 bit representation of $m \mid (e \ll 23) = m + e \times 2^{23}$.

Then we also have the approximation that $\log_2(1+x) \approx x + \mu$ for small values of x, for which our mantissa m is always between zero and one (note, implicit plus one because of the implicit leading one).

So for some floating point value x we have

$$\begin{aligned}
\log_2(x) &= \log_2\left(\left(1 + \frac{m}{2^{23}}\right) \times 2^{e-127}\right) \\
&= \log_2\left(1 + \frac{m}{2^{23}}\right) + \log_2\left(2^{e-127}\right) \\
&= \log_2\left(1 + \frac{m}{2^{23}}\right) + e - 127 \\
&\approx \frac{m}{2^{23}} + e - 127 + \mu \qquad \text{as } \frac{m}{2^{23}} \text{ is small} \\
&= \frac{m + e \times 2^{23}}{2^{23}} + \mu - 127 \\
&= \frac{1}{2^{23}} \text{IEEE 754 bits of } x + \mu - 127
\end{aligned}$$

Now we want some $y = \frac{1}{\sqrt{x}}$, so

$$\begin{aligned}
\log_2(y) &= \log_2\left(\frac{1}{\sqrt{x}}\right) \\
&= -\frac{1}{2}\log_2(x)
\end{aligned}$$

$$\begin{aligned}
\Rightarrow \frac{1}{2^{23}}\text{bits of } y + \mu - 127 &= -\frac{1}{2}\left(\frac{1}{2^{23}}\text{bits of } x + \mu - 127\right) \\
\Rightarrow \frac{1}{2^{23}}\text{bits of } y &= 127 - \mu - \frac{1}{2}\left(\frac{1}{2^{23}}\text{bits of } x + \mu - 127\right) \\
\Rightarrow \text{bits of } y &= 2^{23}(127 - \mu) - \frac{1}{2}\left(\text{bits of } x + 2^{23}(\mu - 127)\right) \\
&= 2^{23}(127 - \mu) + \frac{1}{2}\left(2^{23}(127 - \mu) - \text{bits of } x\right) \\
&= 2^{23} \times \frac{3}{2}(127 - \mu) - \frac{1}{2}\text{bits of } x
\end{aligned}$$

Solving 0x5f3759df $= 2^{23}\frac{3}{2}(127 - \mu)$ for μ gives us $\mu = 0.04504656791687012$ which isn't quite the lowest average error value of 0.043 but is close enough. Apparently this was found via trial and error.

2. JavaScript uses IEEE 754 double-precision floating point (1 bit of sign, 11 bits of exponent, and 52 bits of mantissa [remember the implicit `1.` so it is 53 bits of significand]). What is the range of integers which can be represented contiguously (without gaps)?

[small]

One gotcha is the implicit one, but keeping that in mind, let's consider the case when all the bits in the mantissa are set, and the exponent makes it an integer. We have

$$1.1111111111111111111111111111111111111111111111111111_2 \times 2^{52}$$

to which we can add 1 to get

$$1.0000000000000000000000000000000000000000000000000000_2 \times 2^{53}$$

but then we cannot add 1 again because that would be represented as

$$\begin{aligned} & 1.0000000000000000000000000000000000000000000000000000_2 \times 2^{53} \\ & +1.0000000000000000000000000000000000000000000000000000_2 \times 2^{0} \\ & = 1.00000000000000000000000000000000000000000000000000001_2 \times 2^{53} \\ \text{rounds to } & 1.0000000000000000000000000000000000000000000000000000 \times 2^{53} \end{aligned}$$

but the result cannot be represented using double-precision floating point, so it gets rounded down.

Similar argument applies to negatives, only the sign bit is different. So we get the range $[-2^{53}, 2^{53}]$.

To double-check that we haven't made an off-by-one error, I usually consider a simplified example, in this case consider a floating-point representation with one bit of mantissa

$$(-1)^{\text{sign}} \times 1.m_0 \times 2^{\text{exp}}$$

Here the smallest integer with all mantissa set to 1 is $1.1_2 * 2 = 11_2 = 3_{10}$, or $2^2 - 1$. Similarly, if we had two bits of mantissa, we would have 7_{10} or $2^3 - 1$. So for 52 bits of mantissa, we have $2^{53} - 1$, so the range is indeed $[-2^{53}, 2^{53}]$.

To inspect a floating point number's bytes in OCaml (OCaml floats are double-precision IEEE 754 binary64), you can do (this is beyond the scope of the course):

```ocaml
type parts = { sign: int64; mantissa: int64; exponent: int64 };;
let parts_of_float d =
  let open Int64
  in let bits = bits_of_float d
  in {
       sign     = logand (shift_right_logical bits 63) 1L;
       exponent = logand (shift_right_logical bits 52) 0x7FFL;
       mantissa = logand bits 0xF_FFFF_FFFF_FFFFL
```

```
     };;

(* Format an integer as a single digit, assumes base is bigger than the
integer *)
let sym_of_int n =
  let i = Int64.to_int n
  in if i < 10
     then Char.chr (i + Char.code '0')
     else Char.chr (i - 10 + Char.code 'a');;

(* Formats an int64 to the given base and width, see `mkSeparator` for an
   example of separator *)
let fmt_int n base width separator =
  let open Int64
  in let base = of_int base
  in let rec loop n index =
      if index < width
      then loop (div n base)
                (index + 1) ^
           Char.escaped (sym_of_int (rem n base)) ^
           separator index
      else ""
  in loop n 0;;

(* Separates digits using given separator into `groupSize` groups *)
let mkSeparator separator groupSize n =
  if n != 0 && n mod groupSize == 0 then separator else "";;

(* Example:
fmt_int 0x1234L 16 8 (mkSeparator "_" 4);;
  "0000_1234" *)

(* Euclid's GCD function, for simplifying fractions *)
let rec gcd a b =
  if b = 0L
  then a
  else gcd b (Int64.rem a b);;

(* Print a simplified fraction *)
let fmt_frac nom denom =
  if nom = 0L
  then "0"
  else let cd = gcd nom denom
       in (Int64.to_string (Int64.div nom cd)) ^ "/" ^
          (Int64.to_string (Int64.div denom cd));;
```

```

(* Format a raw double as decimal sign, fraction, exponent; hexadecimal bits and
   binary bits *)
let fmt_parts ({sign = s; mantissa = m; exponent = e}) =
  let open Int64
  in let decfmt = (if s = 0L then " " else "-") ^
                  "(1 + " ^ (fmt_frac m (shift_left 1L 52)) ^
                  ") * 2^" ^ (to_string (sub e 1023L))
  in let hexfmt = "S " ^ fmt_int s 16 1 (fun _ -> "") ^
                  " E " ^ fmt_int e 16 3 (mkSeparator "  " 4) ^
                  " M " ^ fmt_int m 16 9 (mkSeparator "  " 4)
  in let binfmt = "S " ^ fmt_int s 2 1 (fun _ -> "") ^
                  " E " ^ fmt_int e 2 8 (mkSeparator "  " 8) ^
                  " M " ^ fmt_int m 2 52 (mkSeparator "  " 8)
  in decfmt ^ "\n" ^ hexfmt ^ "\n" ^ binfmt;;
let fmt_double d = fmt_parts @@ parts_of_float d;;

print_string @@ fmt_double 1.0 ^ "\n";;
(*  (1 + 0) * 2^0
   S 0 E 3ff M 0  0000  0000
   S 0 E 11111111 M 0000  00000000  00000000  00000000  00000000  00000000  00000000 *)

print_string @@ fmt_double 1.5 ^ "\n";;
(*  (1 + 1/2) * 2^0
   S 0 E 3ff M 0  0000  0000
   S 0 E 11111111 M 1000  00000000  00000000  00000000  00000000  00000000  00000000 *)

print_string @@ fmt_double 1.25 ^ "\n";;
(*  (1 + 1/4) * 2^0
   S 0 E 3ff M 0  0000  0000
   S 0 E 11111111 M 0100  00000000  00000000  00000000  00000000  00000000  00000000 *)

print_string @@ fmt_double (-1.25) ^ "\n";;
(* -(1 + 1/4) * 2^0
   S 1 E 3ff M 0  0000  0000
   S 1 E 11111111 M 0100  00000000  00000000  00000000  00000000  00000000  00000000 *)

print_string @@ fmt_double 0.5 ^ "\n";;
(*  (1 + 0) * 2^-1
   S 0 E 3fe M 0  0000  0000
```

```
    S 0 E 11111110 M 0000  00000000  00000000  00000000  00000000  00000000  00000000 *)

print_string @@ fmt_double 0.25 ^ "\n";;
(*  (1 + 0) * 2^-2
    S 0 E 3fd M 0  0000  0000
    S 0 E 11111101 M 0000  00000000  00000000  00000000  00000000  00000000  00000000 *)

print_string @@ fmt_double @@ float_of_int @@ Int.shift_left 1 53;;
(*  (1 + 0) * 2^53
    S 0 E 434 M 0  0000  0000
    S 0 E 00110100 M 0000  00000000  00000000  00000000  00000000  00000000  00000000 *)

print_string @@ fmt_double @@ float_of_int @@ (Int.shift_left 1 53) - 1;;
(*  (1 + 4503599627370495/4503599627370496) * 2^52
    S 0 E 433 M f  ffff  ffff
    S 0 E 00110011 M 1111  11111111  11111111  11111111  11111111  11111111  11111111 *)
```

3. Solve exercise 1.6 of the lecture notes (copied for convenience).

   Another example of the inaccuracy of floating-point arithmetic takes the golden ration $\varphi \approx 1.618$ as its starting point:

$$\gamma_0 = \frac{1+\sqrt{5}}{2} \quad \text{and} \quad \gamma_{n+1} = \frac{1}{\gamma_n - 1}$$

   In theory it is easy to prove that $\gamma_n = \ldots = \gamma_1 = \gamma_0$ for all $n > 0$. Code this computation in OCaml and report the value of γ_{50}. *Hint:* in OCaml, `sqrt 5` is expressed as `sqrt 5.0`.

[small]

The only gotcha is using the floating point functions `+.`, `-.`, `/.` and floating point literals.

```ocaml
let rec gamma = function
  | 0 -> (1. +. sqrt 5.) /. 2.
  | n -> 1. /. (gamma (n - 1) -. 1.) ;;

gamma 50 ;; (* Evaluates to -0.618121843485747391 *)
```

We can verify that the positive root of $\gamma^2 - \gamma - 1 = 0$ is unstable, while the negative root is stable, so it initially oscillates until it escapes the positive root, then it settles on the negative root.

```ocaml
let (_, _, l) =
  List.fold_left
    (function (n, prev, ls) -> fun l ->
```

```
          if Float.abs @@ prev -. l > 0.1
          then (n+1, l, (n, l)::ls)
          else (n+1, prev, ls))
        (0, 0.0, [])
        @@ List.init 1000 gamma
  in List.rev l;;

  (* [(0, 1.6180339887498949); (37, 1.42633592659499531);
      (38, 2.34556821890819922); (39, 0.743180454136621149);
      (40, -3.89378462857329932); (41, -0.204340827375464856);
      (42, -0.830329734963174526); (43, -0.546349644491854081);
      (44, -0.646684275811768239)] *)
```

2. Complexity

1. Solve one of the following equations:

$$T(1) = 1 \qquad T(n) = 2T\left(\frac{n}{2}\right) + 1$$

or

$$T(1) = 1 \qquad T(n) = T\left(\frac{n}{2}\right) + n$$

[medium]

The trick is repeated expansion, they are pretty much the same

$$T(n) = 2T\left(\frac{n}{2}\right) + 1 \qquad\qquad T(n) = T\left(\frac{n}{2}\right) + n$$

let $2^m = n$

$$\begin{aligned} T(2^m) &= 2T(2^{m-1}) + 1 \\ &= 2(T(2^{m-2}) + 1) + 1 \\ &= \underbrace{2(2(\cdots 2(2^0) + 1 \cdots) + 1) + 1}_{m \text{ expansions}} \\ &= 2^m + 2^{m-1} + \cdots + 2^1 + 2^0 \\ &= 2^{m+1} - 1 \\ T(n) &= 2n - 1 \end{aligned}$$

$$\begin{aligned} T(2^m) &= T(2^{m-1}) + 2^m \\ &= T(2^{m-2}) + 2^{m-1} + 2^m \\ &= \underbrace{1 + 2 + \cdots + 2^{m-1} + 2^m}_{m \text{ expansions}} \\ &= 2^{m+1} - 1 \\ T(n) &= 2n - 1 \end{aligned}$$

To see that $2^m + \cdots + 2 + 1 = 2^{m+1} - 1$, I visualise the sequence $2^m + \cdots + 2 + 1$ as the m bit binary number with all ones $1 \cdots 11$. If I add and subtract one, it is still the same number, but $1 \cdots 11 + 1 - 1 = 10 \cdots 00 - 1$. Alternatively you can just use the geometric sum formula:

$$S_m = ab^0 + ab^1 + \cdots + ab^m = \frac{a(1 - b^{m+1})}{1 - b}$$

2. The Fibonacci function can be written as

```ocaml
let rec fib(n) =
    if n<2 then 1
    else fib(n-2) + fib(n-1) ;;
```

What is its time and space complexity?

[small]

We have $T(0) = 1, T(1) = 1$ and $T(n) = T(n-2) + T(n-1)$ otherwise. This is exactly the Fibonacci series, so $T(n) = \text{fib}(n) = O(\varphi^n)$ since $\varphi^n = \text{fib}(n) \times \varphi + \text{fib}(n-1)$.

For space complexity, it is linear.

3. The Fibonacci function can be more efficiently written if instead of returning a single value of `fib(n)`, we return two values, `(fib(n), fib(n-1))`. Write this function.

   What is its time and space complexity?

[small]

This is actually simpler, as it is just linear time, linear space.

```ocaml
let rec fib' n =
    if n<2 then (1, 1)
    else let (n1, n2) = fib'(n-1)
          in (n1+n2, n1) ;;
```

To get the intended output, we need to wrap it, though this isn't the interesting part of the question.

```ocaml
let fib n = let (fn, _) = fib' n in n;;
```

You can check that this isn't tail-recursive, by inserting a tail-call annotation (OCaml always does tail-call optimisation if possible, but the annotation will print a warning if it is not possible).

```ocaml
let rec fib' n =
    if n<2 then (1, 1)
    else let (n1, n2) = (fib'[@tailcall])(n-1)
          in (n1+n2, n1) ;;
```

This gives us the warning:

```
Warning 51 [wrong-tailcall-expectation]: expected tailcall
```

It is possible to make a tail-recursive Fibonacci function in OCaml however, it is very similar to just using a for-loop in an imperative language:

```ocaml
let rec fib''loop(n1, n2, index, limit) =
  if index < limit
  then (fib''loop[@tailcall]) (n1+n2, n1, index+1, limit)
  else n1;;
let rec fib'' n = fib''loop(1, 1, 1, n);;
```

Which has no such warning.

3. Lists and recursion

1. What is the difference between `List.fold_left` and `List.fold_right`? Write your own version of them (say `foldl` and `foldr`) using pattern-matching on the list.

[small]

The difference is the associativity, and the space-complexity.

$$\begin{aligned} \operatorname{foldl} \operatorname{op} \operatorname{acc} \; [x_0, ..., x_n] &= ((\operatorname{acc} \operatorname{op} x_0) \operatorname{op} \cdots) \operatorname{op} x_n && \text{infix} \\ &= \operatorname{op}(\operatorname{op}(... \operatorname{op}(\operatorname{acc}, x_0)...), x_n) && \text{prefix} \\ \operatorname{foldr} \operatorname{op} [x_0, ..., x_n] \; \operatorname{acc} &= x_0 \operatorname{op}(\cdots \operatorname{op}(x_n \operatorname{op} \operatorname{acc})) && \text{infix} \\ &= \operatorname{op}(x_0, \operatorname{op}(... \operatorname{op}(x_n, \operatorname{acc})...)) && \text{prefix} \end{aligned}$$

```ocaml
(* val foldl : ('acc -> 'elem -> 'acc) -> 'acc -> 'elem list -> 'acc ;; *)
let rec foldl f acc list = match list with
  | []    -> acc
  | l::ls -> foldl f (f acc l) ls ;;

(* val foldr : ('elem -> 'acc -> 'acc) -> 'elem list -> 'acc -> 'acc ;; *)
let rec foldr f list acc = match list with
  | []    -> acc
  | l::ls -> f l (foldr f ls acc) ;;
```

As it can be seen, fold-left is tail-recursive (so constant space), but fold-right is not (so linear space). Both are linear time in the length of the list.

2. List folding is the principal function for operating on lists, all other functionality can be implemented on top of `fold_left`. Using `fold_left`, implement `rev l` for reversing the list `l`, and `map f l` for applying `f` to each element of the list `l`.

[small]

```ocaml
let rev l = foldl (fun ls l -> l::ls) [] l ;;

let map f l = rev (foldl (fun ls l -> (f l)::ls) [] l) ;;

(* Or *)
let map f l = foldr (fun l ls -> (f l)::ls) l [] ;;
```

3. Write a function which takes a list of strings and formats it, for example `fmtlist [1;2;3]` evaluates to `"[1; 2; 3]"`. You might want to write this function in terms of `foldl` to help with the next part. Note, string concatenation is done using the ^ operator in OCaml.

[small]

```ocaml
let fmtlist l =
```

```ocaml
  let (str, _) = foldl (fun (str, sep) s -> (str ^ sep ^ s, "; ")) ("", "") l
  in "[" ^ str ^ "]"
;;
```

4. Fold-left is such a fundamental operation of lists, that lists can be encoded as a function that represents fold-left. (Can be encoded, but not encoded this way usually.) For example,

```ocaml
let l     (* encoding [] *)          = fun f v -> v                ;;
let l_1   (* encoding ["1"] *)       = fun f v -> f v "1"          ;;
let l_1_2 (* encoding ["1"; "2"] *) = fun f v -> f (f v "1") "2" ;;

let foldl f v l = l f v ;;

(* Copy/re-evaluate your `fmtlist` to use the new fold *)
fmtlist l     ;; (* Evaluates to "[]" *)
fmtlist l_1   ;; (* Evaluates to "[1]" *)
fmtlist l_1_2 ;; (* Evaluates to "[1; 2]" *)
```

Can you come up with a function that conses an element to a list, in this representation? So that

```ocaml
fmtlist (cons "0" l_1_2) ;; (* Evaluates to "[0; 1; 2]" *)
```

[big, if you can't figure it out it's okay]

What we want to do is for a list like `ocaml fun f v -> f (f v "1") "2"` put the expression `ocaml (f' v' "0")` in place of `v`, so we want to apply that to `v`. We also need to replace `f` with `f'`, so what we do is:

```ocaml
let cons l ls = fun f v -> ls f (f v l) ;;
```

Almost correct is appending to the end:

```ocaml
let cons' l ls = fun f v -> f (ls f v) l ;;
fmtlist (cons' "0" l_1_2) ;; (* Evaluates to "[1; 2; 0]" *)
```