Je n'ai pas encore d'exos ici, car c'est beaucoup d'introduction aux bases de l'impératif et au cours sur le fonctionnement de la machine, et les exos y sont rarement intéressants. On pourrait quand même faire l'analyse de la fast inverse square-root ou des bizarreries avec des flottants.

Cours

- Paradigmes: impératif structuré, déclaratif fonctionnel, logique
- Savoir coder en C
- Différence entre compilé et interprété
- Représentation des entiers: signé, non signé, bytes/bits
- Représentation des flottants: signe, mantisse, exposant
- Preuve sur programme. Invariant de boucle. Correction partielle (vrai si termine), correction totale (vrai et termine)
- Complexité : pire cas, cas moyen, coût amorti
- Spécialisation d'une fonction. Commentaires, annotation
- Programmation défensive, assertions
- Jeu de test. Graphe de flot de contreole. Chemin faisable. Couverture des sommets, couverture des arêtes. Test exaustif d'une boucle.
- Pointeurs, allocation, stack, heap, call stack, stack overflow.

Complexité

En complexité il y a 2 écoles: celle qui suppose que les opérations sur les entiers (addition, multiplication, xor) sont en $O(1)$ et celle de ceux qui pensent que c'est en $O(\log n)$. On cherche maintenant à analyser la complexité dans la seconde école, avec des entiers de taille arbitraire. Pour ce faire, on représentera un entier par un `type int' = bool list;` tel que l'entier soit représenté en base 2 **à l'envers** avec (`true` = 1, `false` = 0).

Par exemple, 19 sera représenté par `[true, true, false, false, true];;`

1. Donner la liste qui encode les nombres 0, 1 et 13
2. Proposer le code de `val incr: int' -> int'` qui incrémente un entier de 1. Pourquoi représenter un entier à l'envers ?

On supposera la fonction `val decr: int' -> int'` écrite.

3. Pour deux entiers A, B, quelle est la complexité du code suivant ? Proposer un code plus rapide. Quelle est la complexité dans le pire des cas ?

```
let rec add (a: int') (b:int'): int' =
  match a with
  | [] | [false] -> b
  | _ -> incr (add (decr a) b)
```

4. Justifier que le programme suivant est bien en $O(N)$

```
let rec etrange (n:int') = match n with
  | [] | [false] -> [false]
  | _ -> incr (etrange (decr n))
```