ULTIME COACHING

1 Généralités

1. Atention aux “évidences” : faire en fonction du contexte de l’épreuve !
2. C’est très important de s’adapter aux notations de l’énoncé.
3. Attention aux récurrences : même quand elle est "immédiate", donner très rapidement les arguments qui font que ça marche (initialisation, hérédité)
4. Attention aux majorations/minorations.
5. Si le sujet veut mieux que “du O(..)”, être un peu soigneux sur les calculs.
6. Les double-implications sont en général plus légères à lire que les équivalences, souvent "plus correctes", et rarement beaucoup plus longues.
7. Comme pour la presque totalité des épreuves de concours, une lecture rapide de l’énoncé peut vous faire comprendre l’esprit global et les objectifs, vous évitant éventuellement de raconter des énormités (une complexité en $\ln n$ pour l’algorithme naïf, là où l’énoncé établit péniblement un algorithme en n^2...).
8. Encadrer les résultats peut être une bonne idée pour soulager le pauvre correcteur, qui verra directement quelle est votre réponse !

2 Automates, langages

1. Pas d’ε-transition, a priori (hors-programme).
2. La plupart des preuves se font par récurrence sur la longueur des chemins/mots. Pensez à faire des dessins, et à traiter les cas particuliers.
3. On a souvent besoin de supposer l’automate déterministe complet, dans les preuves; pensez-y !
4. Savoir déterminiser calmement un automates (méthode des parties). Au fait quel est le coût de cette construction ?
5. Savoir construire un automate reconnaissant un langage donné par une expression rationelle.
6. Connaître le lemme de l’étoile. Savoir rédiger correctement la méthode de séparation d’états (c’est **toujours** la même chose). C’est pas vraiment au programme, mais c’est bien quand même.
7. Faire des dessins... mais ne pas s’en contenter.

3 Programmation

1. Vérifier rapidement le type de chaque fonction, voire le mettre sur la copie.
2. Ne pas mélanger itératif (impératif) et récursif (fonctionnel). C’est parfois imposé dans l’énoncé.
3. Les confusions listes/vecteurs sont LE mal absolu.
4. Penser simple (récursif, la plupart du temps) et jouer simple (matching élémentaire).
5. Ne pas matcher sur les vecteurs !
6. Ne pas matcher avec des noms de variables déjà utilisés en espérant que ça matchera la même valeur...
7. Ne pas abuser, si possible, des "`when`" dans les pattern matching. Ils sont parfois élégants, mais aussi régulièrement d’une grandeur lourdeur/laideur, et pourraient être remplacés par de vulgaires mais efficaces "`if ... then ... else .`

8. Expliquer voire typer les fonctions auxiliaires (éventuellement extérieures : c’est souvent plus clair ainsi) que vous écrivez.
9. Réfléchir au programme **avant** de porcifier sa copie avec des ratures. Un premier jet au brouillon est même envisageable...
10. Pour les listes, penser au matching simple (`[]` et `x::reste`, qui suffisent la plupart du temps). Bien distinguer la concaténation `@` et l’adjonction `::` Si on doit concaténer, signaler qu’on sait que c’est très mal, mais que vu le contexte et les autres complexités en jeu, et bien voila, c’est comme ça et pas autrement !
11. Indenter, et sur-parenthéser (ou `begin ... end`) pour éviter toute ambiguïté ou foulage de cerveau du correcteur.
12. Éviter les cascades à bases de `List.iter, List.map, List.fold_left` ou `raise(...)` sauf si c’est **vraiment** maîtrisé. Et encore, cela pourra être vu comme un contournement de la difficulté, dans certains cas. Pour `map` il convient de sentir si son utilisation serait un détournement de l’esprit du sujet. A priori, si ce n’est pas explicitement interdit, on peut considérer que c’est probablement autorisé !
13. Ne pas faire deux fois un appel à une fonction sur la même entrée ; nommer le résultat (avec `let ... in`). La complexité peut en être complètement modifiée !
14. Sans aller jusqu’à un code forcément exécutable, les erreurs de syntaxe trop nombreuses (oublis de ";", "done" et cie, finissent par piquer les yeux des correcteurs.
15. Vous devez savoir, les yeux bandés :
    (a) Écrire une fonction `rev` linéaire.
    (b) Mettre dans une liste les éléments d’un arbre binaire en temps linéaire (sans concaténation).
    (c) Coder un tri par insertion, sélection, le tri-fusion, et le tri-rapide. Sur un vecteur comme sur une liste, et les mains dans le dos (cf. première année).
    (d) Idem pour les parcours d’arbre, de graphe, ...
16. Avant les écrits, faire une petite piqure de rappel concernant la syntaxe des chaînes de caractères et structure d’enregistrement :
    `type arbre={etiq:int; fils:arbre liste};`
17. Les "`let x::reste=une_liste in...`" sont, je trouve, très laids. Un matching sur `une_liste` pourra la plupart du temps le remplacer de façon élégante.
18. On a bien rigolé une ou deux fois avec les `if gnagna=true then...`, mais maintenant, fin de la récréation.
19. Heu, `if gnagna then true else false`... ben pareil.
20. Au fait, `if gnagna then false else true`, ce n’est pas très différent.
21. Non mais **vraiment**.
22. Évitez de traiter cinquante cas de bases. Exceptionnellement, on peut faire un traitement redondant pour clarifier le fonctionnement (le préciser alors).
23. Vous pouvez commenter votre code aussi (un peu)...
24. Les "`failwith ''n'arrive pas'';`" peuvent alléger la lecture d’un matching non exhaustif.

4 Algorithmique, complexité

1. Dans les preuves d’algorithmes/programmes, chercher le bon invariant.
2. Attention aux *"Le coût est minimal lorsque* `{insert whatever here}`*"*.
3. Bien maîtriser les trois ou quatre situations de base : $C(n+1) = C(n) + O(n^p)$, $C(n) = 2C(n/2) + O(n)$, $C(n) = 3C(n/2) + O(n)$...
4. Sauf mention explicite, on demande les complexités dans le pire des cas. Une petite remarque sur la complexité moyenne prévisible peut être la bienvenue, mais ne vous dispersez pas !
5. Réfléchir à la complexité des calculs arithmétiques de base. Ça peut dépendre de si les valeurs manipulées tiennent dans un mot mémoire de l’ordinateur (32 bits, 64 bits)

5 Arbres

1. Faire des dessins.
2. Les preuves se font presque systématiquemnt par récurrence sur la hauteur ou le nombre de nœuds. Ou par induction.
3. Les "La hauteur est minimale (resp. maximal) lorsque `{insert whatever here}`" sont en général à éviter (ça se prête souvent à du pipotage).
4. Penser que $\ln_2 n \leq h \leq n-1$, et que l'objectif est d'avoir $h = O(\ln n)$.
5. Il faut être capable d'établir les résultats combinatoires simples (nombre de nœuds en fonction de la hauteur...). Une récurrence suffit, la plupart du temps.
6. La plupart des algorithmes demandent une seule descente dans l'arbre; éviter les parcours exhaustifs !
7. Faire des dessins propres.
8. Codez simplement, comme pour les listes. La plupart du temps, les fonctions de bases se codent avec un matching simple sur l'arbre vide d'une part, et un arbre du genre `N(fg,etiq,fd)` d'autre part. Ne faites autre chose que si le matching élémentaire ne marche pas.
9. Vous devez pouvoir retrouver (sans le cours) les algorithmes de base sur les ABR par exemple (recherche, insertion à la racine ou aux feuille, suppression). (sans équilibrage)
10. Faire des dessins propres et jolis.

6 Les grosses erreurs

1. Ne pas confondre Python et OCaml ! N'inventez pas de `return` ou équivalent !
2. Il faut justifier, justifier, justifier. Surtout pour les questions faciles au début.
3. Vous pouvez justifier succintement si vos arguments sont bons. Pas besoin de 3 pages pour répondre à une seule question.

Des trucs en plus

Lemme d'Arden (HP)

Soit A et B deux langages sur un même alphabet.

1. L'équation $L = AL \cup B$ (d'inconnue L) admet $L = A^*B$ comme solution.
2. $L = A^*B$ est la plus petite solution (au sens de l'inclusion)
3. Si $\varepsilon \notin A$, alors $L = A^*B$ est la seule solution. (Sinon Σ^* est solution par exemple)

Preuves :

1. A^*B est solution car $AA^*B \cup B = (AA^* \cup \{\varepsilon\})B = A^*B$
2. Si L est une solution alors, on a par récurrence sur $n \in \mathbb{N}$,
$$L = A^{n+1}L \cup (\cup_{k=0}^{n} A^k B)$$
Donc pour tout $n \in \mathbb{N}$, $A^nB \subset L$. Donc $A^*B \subset L$
3. Si $\varepsilon \notin A$, considérons x un mot qui fait partie d'une solution L de longueur n. On a encore $L = A^{n+1}L \cup (\cup_{k=0}^{n} A^k B)$ or les mots de A^{n+1} sont de longueur au moins $n+1$ car $\varepsilon \notin L$. D'où $x \in \cup_{k=0}^{n} A^k B)$ et donc $L \subset A^*B$

Essayer sur des exemples : commencer par des exemples très très très simples (avec 2 états) : que fait-on des états initiaux ? terminaux ?

Floyd-Warshall (Mathieu Roux)

Pourquoi n'y a-t-il pas besoin de faire une copie de la matrice avant de la modifier? En effet, quid si la case $m_{i,k}^{(k)}$ est modifiée et vaut $m_{i,k}^{(k+1)}$ au moment de calculer $m_{i,j}^{(k+1)}$?

En fait, les formules
$$m_{i,j}^{(k+1)} = \min\left(m_{i,j}^{(k)}, m_{i,k}^{(k+1)} + m_{k,j}^{(k)}\right),$$
$$m_{i,j}^{(k+1)} = \min\left(m_{i,j}^{(k)}, m_{i,k}^{(k)} + m_{k,j}^{(k+1)}\right),$$
$$m_{i,j}^{(k+1)} = \min\left(m_{i,j}^{(k)}, m_{i,k}^{(k+1)} + m_{k,j}^{(k+1)}\right),$$
sont encore valables. En effet, en l'absence de cycle de poids négatif, $m_{i,k}^{(k+1)} = m_{i,k}^{(k)}$ et $m_{k,j}^{(k+1)} = m_{k,j}^{(k)}$, donc on peut bien mettre les coefficients de la matrice à jour dans l'ordre que l'on veut.

Algorithmes de tris

OCaml : Tri par insertion, Tri fusion Tri rapide

voir pages suivantes (Jean-Pierre Becirspahic, Lycée Louis-le-Grand)

Langages locaux

Stabilité des langages locaux par les opérations usuelles (intersection, étoile...) : certains de ces résultats ne sont vrais que si les deux langages sont sur des alphabets disjoints.

voir pages suivantes (Jules Svartz, Lycée Masséna)

Travaux pratiques

option informatique

Tri d'une liste

Durant cette séance nous allons programmer trois algorithmes de tri classiques sous la forme de fonctions opérant sur des listes.

Exercice 1. tri par insertion

a) Écrire une fonction **insertion** qui insère un élément dans une liste dont les éléments sont supposés *rangés par ordre croissant*.

```
insertion : 'a -> 'a list -> 'a list
```

b) Le tri par insertion consiste à extraire le premier élément de la liste, à trier récursivement le reste de la liste, puis à insérer l'élément isolé à l'aide de la fonction précédente.
Écrire une fonction **insertion_sort** qui réalise le tri par insertion d'une liste.

```
insertion_sort : 'a list -> 'a list
```

c) Justifier que le coût de cette fonction est un $O(n^2)$ où n désigne la taille de la liste à trier.

Exercice 2. tri fusion

a) Écrire une fonction **split** qui prend une liste ℓ en argument et renvoie une paire de listes (ℓ_1,ℓ_2) telle que les éléments de ℓ sont exactement ceux de ℓ_1 réunis à ceux de ℓ_2 et telle que les longueurs de ℓ_1 et de ℓ_2 diffèrent d'au plus une unité.

```
split : 'a list -> 'a list * 'a list
```

b) Ecrire une fonction **merge** qui prend en arguments deux listes supposées triées par ordre croissant et les fusionne en une unique liste triée.

```
merge : 'a list -> 'a list -> 'a list
```

c) Le tri fusion consiste à couper la liste en deux parties à l'aide de la fonction **split**, à trier récursivement chacune des deux listes obtenues puis à les fusionner à l'aide de la fonction **merge**.
Ecrire une fonction **mergesort** qui réalise le tri fusion d'une liste.

```
mergesort : 'a list -> 'a list
```

Nous démontrerons plus tard dans l'année que le coût de cette fonction est un $O(n\log n)$ où n désigne la taille de la liste à trier.

Exercice 3. tri rapide

a) Ecrire une fonction **partition** qui prend en arguments un élément x et une liste ℓ et qui renvoie une paire de listes (ℓ_1,ℓ_2) telle que ℓ_1 contient les éléments de ℓ qui sont inférieurs ou égaux à x et ℓ_2 ceux qui lui sont strictement supérieurs.

```
partition : 'a -> 'a list -> 'a list * 'a list
```

b) Le tri rapide consiste à isoler un élément particulier de la liste (par exemple le premier), à partitionner les éléments restants à l'aide de la fonction précédente, à trier récursivement les deux listes obtenues, puis enfin à « recoller » les morceaux (par concaténation).
Écrire une fonction **quicksort** qui réalise le tri rapide d'une liste.

```
quicksort : 'a list -> 'a list
```

Nous démontrerons plus tard que le coût de cette fonction est dans le pire des cas un $O(n^2)$ mais qu'en moyenne son coût est un $O(n\log n)$.

Corrigé

option informatique

Tri d'une liste

Exercice 1. tri par insertion

```
let rec insere x = function
  | []              -> [x]
  | t::q when t < x -> t::(insere x q)
  | l               -> x::l ;;

let rec insertion_sort = function
  | []   -> []
  | [a]  -> [a]
  | t::q -> insere t (insertion_sort q) ;;
```

Exercice 2. tri fusion

```
let rec split = function
  | []      -> [], []
  | [a]     -> [a], []
  | a::b::q -> let l1, l2 = split q in a::l1, b::l2 ;;

let rec merge l1 l2 = match (l1, l2) with
  | [], _                       -> l2
  | _, []                       -> l1
  | t1::q1, t2::_ when t1 < t2 -> t1::(merge q1 l2)
  | _, t2::q2                   -> t2::(merge l1 q2) ;;

let rec mergesort = function
  | []  -> []
  | [a] -> [a]
  | l   -> let (l1, l2) = split l in
           merge (mergesort l1) (mergesort l2) ;;
```

Exercice 3. tri rapide

```
let rec partition x = function
  | []               -> [], []
  | t::q when t <= x -> let l1, l2 = partition x q in t::l1, l2
  | t::q             -> let l1, l2 = partition x q in l1, t::l2 ;;

let rec quicksort = function
  | []   -> []
  | [a]  -> [a]
  | t::q -> let l1, l2 = partition t q in (quicksort l1) @ (t::(quicksort l2)) ;;
```

Propriétés de clôture des langages locaux

L'ensemble des langages locaux est stable par certaines opérations. Voyons comment construire des langages locaux à partir d'autres langages locaux. Dans la suite, on fera des opérations entre deux langages L_1 et L_2, supposés locaux. On note pour $i \in \{1,2\}$ P_i, S_i, F_i et N_i les ensembles $P(L_i)$, $S(L_i)$, $F(L_i)$ et $N(L_i)$.

a) Intersection

Proposition 13.47. *L'intersection de deux langages locaux est local.*

Démonstration. On considère deux langages L_1 et L_2 locaux. Si $m \neq \varepsilon$ est dans $L_1 \cap L_2$, sa première lettre est dans $P_1 \cap P_2$, ses facteurs sont dans $F_1 \cap F_2$, et sa dernière lettre est dans $S_1 \cap S_2$. La réciproque est immédiate, donc avec $N = \Sigma^2 \backslash (F_1 \cap F_2)$.

$$L \backslash \{\varepsilon\} = (P_1 \cap P_2)\Sigma^* \cap \Sigma^*(S_1 \cap S_2) \backslash \Sigma^* N \Sigma^*$$

La propriété 13.45 montre que L est local. □

b) Union et Concaténation sur alphabets disjoints

Les langages locaux ne sont pas stables par union et concaténation, par contre si L_1 et L_2 sont locaux sur alphabets disjoints, c'est le cas.

Proposition 13.48. *Si L_1 et L_2 sont deux langages locaux sur Σ_1 et Σ_2 avec $\Sigma_1 \cap \Sigma_2 = \emptyset$, alors $L = L_1 \cup L_2$ est local sur $\Sigma = \Sigma_1 \cup \Sigma_2$.*

Démonstration. Posons $P = P_1 \cup P_2$, $S = S_1 \cup S_2$ et $F = F_1 \cup F_2$, et $N = \Sigma^2 \backslash F$. Ce sont clairement les ensembles caractéristiques de L, donc $L \backslash \{\varepsilon\} \subset P\Sigma^* \cap \Sigma^* S \backslash \Sigma^* N \Sigma^*$. Soit $m \neq \varepsilon$ un mot de $P\Sigma^* \cap \Sigma^* S \backslash \Sigma^* N \Sigma^*$. Si la première lettre de m est dans P_1, on montre de proche en proche que tous les facteurs de taille 2 sont dans F_1, et sa dernière lettre dans S_1. Donc $m \in L_1 \subset L_1 \cup L_2$. De même si la première lettre est dans P_2. Ainsi $L_1 \cup L_2$ est local. □

Remarque 13.49. *Si $\Sigma_1 \cap \Sigma_2 \neq \emptyset$, la propriété est fausse : par exemple $\{ab\}$ et $\{bc\}$ sont tous deux locaux mais pas leur union, qui devrait contenir abc.*

Proposition 13.50. *Si L_1 et L_2 sont deux langages locaux sur Σ_1 et Σ_2 avec $\Sigma_1 \cap \Sigma_2 = \emptyset$, alors $L_1 \cdot L_2$ est local sur $\Sigma = \Sigma_1 \cup \Sigma_2$.*

Démonstration. Le tableau suivant donne les ensembles $P(L)$, $S(L)$ et $F(L)$ avec $L = L_1 \cdot L_2$ en fonction des ensembles caractéristiques de L_1 et L_2, dans le cas où les deux langages sont non vides (sinon L est vide, et donc local) :

	$\varepsilon \notin L_2$	$\varepsilon \in L_2$
$\varepsilon \notin L_1$	$P(L) = P_1$ $S(L) = S_2$ $F(L) = F_1 \cup F_2 \cup S_1P_2$	$P(L) = P_1$ $S(L) = S_1 \cup S_2$ $F(L) = F_1 \cup F_2 \cup S_1P_2$
$\varepsilon \in L_1$	$P(L) = P_1 \cup P_2$ $S(L) = S_2$ $F(L) = F_1 \cup F_2 \cup S_1P_2$	$P(L) = P_1 \cup P_2$ $S(L) = S_1 \cup S_2$ $F(L) = F_1 \cup F_2 \cup S_1P_2$

Soit m un mot de $P(L)\Sigma^* \cup \Sigma^* S(L) \backslash \Sigma^* N(L) \Sigma^*$.

— si sa première lettre est dans P_1, considérons m_1 son plus grand préfixe dans Σ_1^*. Ses facteurs de taille 2 sont tous dans Σ_1^2, donc dans F_1.

— si $m_1 = m$ (auquel cas $\varepsilon \in L_2$), alors la dernière lettre de m est dans S_1, donc $m \in L_1 \subseteq L_1L_2$.

— sinon, notons m_2 tel que $m_1m_2 = m$. Le facteur de taille 2 constitué de la dernière lettre de m_1 et la première de m_2 est nécessairement dans S_1P_2, donc $m_1 \in L_1$. On montre de proche en proche que les facteurs suivant sont tous dans Σ_2^2, donc dans F_2. La dernière lettre de m est dans Σ_2, donc dans S_2. Ainsi $m_2 \in L_2$, et $m \in L$.

— sinon, on a $\varepsilon \in L_1$ et on montre facilement que $m \in L_2 \subseteq L_1L_2$.

Ainsi, L est local. □

Remarque 13.51. *Là encore, si $\Sigma_1 \cap \Sigma_2 \neq \emptyset$, la propriété est fausse : par exemple $\{ab\}$ et $\{bc\}$ sont tous deux locaux mais pas leur concaténation $\{abbc\}$, qui devrait contenir abc.*

c) Étoile de Kleene

Proposition 13.52. *Si L est local, alors L^* est local.*

Démonstration. Avec P, S, et F les ensembles caractéristiques de L, ceux de L^* sont : $P(L^*) = P$, $S(L^*) = S$, et $F(L^*) = F \cup S \cdot P$. Montrons que l'on a bien $P(L^*)\Sigma^* \cup \Sigma^* S(L^*) \backslash \Sigma^* N(L^*) \Sigma^* \subset L^* \backslash \{\varepsilon\}$: Considérons un tel mot m, et regardons ses facteurs qui sont dans $S \cdot P \backslash F$: ceux-ci induisent une factorisation de m en mots $m_1 m_2 \cdots m_k$, où la dernière lettre de m_i est dans S, la première lettre de m_i dans P, et tous les facteurs de taille 2 de m_i dans F : donc $m_i \in L$ et $m \in L^*$. □

d) Corollaire

On déduit des points précédents le théorème qui suit :

Théorème 13.53. *Le langage associé à une expression linéaire est un langage local.*

Démonstration. ε et $\emptyset$ sont des langages locaux (les ensembles caractéristiques sont vides). Le langage $L_a = \{a\}$ sur Σ est local, avec $P(L_a) = S(L_a) = \{a\}$ et $F(L_a) = \emptyset$. Ensuite, une expression linéaire est construite comme $e_1 + e_2$ ou e_1e_2 avec e_1 et e_2 deux expressions linéaires sur alphabets disjoints, ou encore comme e^* avec e linéaire. On conclut par induction. □

Remarque 13.54. *La réciproque est fausse : le langage dénoté par aa^* est local, car égal à $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$. En revanche, il n'est pas local. De même pour $L(baa^*c)$, par exemple.*

Remarque 13.55. *Pour $L = L_1L_2$, $L = L_1 \cup L_2$ ou $L = L_1^*$, les expressions donnant $P(L), S(L)$ et $F(L)$, en fonction des $P(L_i), S(L_i)$ et $F(L_i)$ vues dans les démonstrations ci-dessus restent valables, mais pas pour $L = L_1 \cap L_2$.*