

# Concours de programmation UQAM (Hiver 2003) : Stratégies de travail en équipe

## 1 Introduction

Ce document vise à décrire certaines stratégies pouvant être utiles dans le cadre du concours de programmation. Il est traduit et adapté d'un document intitulé "*Teamwork in Programming Contests: 3 \* 1 = 4*"<sup>1</sup>.

Ces diverses stratégies sont toutes motivées par le premier critère devant être satisfait pour remporter un tel concours de programmation, à savoir, résoudre le plus grand nombre de problèmes.

## 2 Les trois facteurs clés

Trois facteurs sont importants pour qu'une équipe obtienne de bons résultats dans un concours de programmation :

1. Avoir une bonne connaissance des algorithmes de base ;
2. Être capable de coder un algorithme dans un langage de programmation ;
3. Avoir une bonne stratégie de travail en équipe.

Concernant le premier facteur, on peut classer les problèmes typiques d'un concours de programmation dans les différentes catégories suivantes :

- Problèmes de fouille et de recherche ;
- Problèmes de graphes ;
- Problèmes géométriques ;
- Problèmes triviaux ;
- Problèmes non-standards.

---

<sup>1</sup><http://www.acm.org/crossroads/xrds3-2/progcon.html>

Il peut donc être utile de réviser au préalable les algorithmes étudiés dans certains cours de base (INF1130, INF2110, INF3102). Notons que des livres de références peuvent aussi être apportés dans les laboratoires lors du concours.

Pour le deuxième facteur (programmation), l'important n'est évidemment pas de taper le plus vite possible ... pour ensuite perdre du temps à *debugger*. Il vaut mieux coder l'algorithme correctement, en pensant dès le départ aux différents cas qui doivent être traités, de façon à s'assurer que le programme fonctionne du premier coup (ou presque), donc en *debuggant* le moins possible.

Le dernier facteur, les stratégies de travail en équipe, est examiné dans les sections qui suivent.

### 3 Principes de base de travail en équipe

Il est important que l'équipe décide au préalable d'une stratégie de travail durant le concours. La raison en est très simple : les membres de l'équipe n'ont droit d'utiliser *qu'une seule* machine, et ce même si leur équipe est formée de trois coéquipiers. Il faut donc diviser intelligemment le travail, quitte à ce que chacun des membres de l'équipe se spécialise dans un aspect du travail. Différentes divisions du travail sont possibles, en fonction des forces et faiblesses des membres de l'équipe : répartition des problèmes entre les membres selon le type de problème, répartition des tâches (analyse du problème, codification, ...), etc. (voir la prochaine section).

Répetons le, l'aspect clé dont il faut tenir compte dans le choix de la stratégie est que l'équipe gagnante est celle qui résout correctement le plus grand nombre de problèmes. Il est donc important que l'équipe essaie de déterminer le mieux possible, avant d'attaquer un problème donné, si la solution de ce problème pourra être complétée dans le temps qui reste. Ainsi, il est souvent utile, au début du concours, de prendre quelques minutes pour faire une séance de remue-méninges (*brain storming*) visant à évaluer la difficulté de chacun des problèmes et à déterminer quels problèmes l'équipe tentera de résoudre en priorité.

La gestion du temps est aussi très importante. Si vous prévoyez attaquer un problème plus difficile, il est généralement préférable de le faire au début du concours. La résolution d'un problème difficile alors qu'il reste peu de temps risque de conduire à une situation où aucun autre problème ne pourra être résolu, faute de temps pour accéder au terminal.

La gestion de l'utilisation du terminal est donc cruciale. Un membre de l'équipe ne devrait être devant le terminal que si il est en train de taper, et non pas s'il est en train de réfléchir. Il est généralement préférable d'écrire le programme sur papier, de la façon la plus détaillée possible, *avant* de s'asseoir devant le terminal pour le taper. Surtout, attention au *debuggage* en ligne, qui peut

prendre un temps énorme et causer un goulot d'étranglement pour l'utilisation du terminal. Si vous rencontrez un *bug*, évitez de perdre du temps à tracer l'exécution de votre programme, surtout si un autre membre de l'équipe a besoin du terminal ; à la place, imprimez votre programme et réfléchissez. Évidemment, s'il reste peu de temps au concours, donc insuffisamment de temps pour amorcer la résolution d'un nouveau problème, alors le *debuggage* peut devenir la priorité.

## 4 Quelques exemples de stratégies

Trois stratégies de base souvent utilisées sont celles décrites dans les sections qui suivent.

### 4.1 Répartition simple des problèmes

Chaque membre travaille individuellement, sans trop d'interactions avec les autres membres. Au début du concours, les équipiers se répartissent les problèmes entre eux. Chacun travaille sur ses différents problèmes, l'un après l'autre.

Un désavantage de cette stratégie est que les problèmes les plus simples risquent d'être tous prêts à être codés en même temps, le terminal devenant ainsi un goulot d'étranglement. Un autre danger : il risque de ne plus rester suffisamment de temps, à la fin, pour résoudre les problèmes plus difficiles.

### 4.2 Le spécialiste du terminal

Dans cette stratégie, un seul membre de l'équipe, appelons-le  $T$ , utilise le terminal. Les deux autres membres de l'équipe analysent les problèmes et écrivent les algorithmes pendant que  $T$  écrit les routines d'E/S appropriées, les jeux d'essai de base (tirés des énoncés de problèmes), puis tape le code et teste le programme. C'est aussi  $T$  qui s'occupe de *debugger* le programme, à moins que le problème soit difficile à identifier.

L'avantage de cette stratégie est que le terminal n'est plus autant un goulot d'étranglement. Par contre, les compétences de l'équipier  $T$  ne sont pas nécessairement utilisées de façon efficace, à moins que  $T$  ne soit l'expert du langage et de la machine utilisée.

### 4.3 Le *think tank*

Dans cette approche, deux membres de l'équipe forment la sous-équipe *think tank* ( $TT$ ). Au début du concours, pendant que les membres de  $TT$  lisent et analysent les problèmes, le troisième membre en profite pour écrire diverses routines de base d'E/S et taper les divers jeux d'essai de base décrits dans les

énoncés de problème. Après dix à quinze minutes de discussion, les membres de *TT* choisissent un premier problème, simple, sur lequel le troisième membre, le programmeur, pourra commencer à travailler aussitôt que les membres de *TT* lui auront expliqué les grandes lignes de la solution. Ensuite, les membres de *TT* continuent, durant environ  $\frac{3}{4}$ -1 heure, à examiner les divers problèmes et à décider lesquels seront traités. Les plus courts et plus simples sont laissés au troisième membre, alors que les autres problèmes sont divisés entre les membres de *TT* (qui s'occuperont aussi de les taper et *debugger*).

Durant les premières heures, lorsqu'un *bug* est rencontré et qu'il ne semble pas pouvoir être résolu rapidement, ou lorsqu'un programme est rejeté par les juges, le programme est laissé de côté pour la dernière heure du concours. Durant cette dernière heure, aucun nouveau code ne devrait être écrit; cette heure devrait plutôt être utilisée pour *debugger* et compléter les programmes qui ne sont pas encore complètement terminés.

## 5 Autres remarques

Comme le dit l'adage, "l'important est de participer". Il ne faut donc pas stresser inutilement, durant le concours, sur le tableau des points, l'état d'avancement des autres équipes, etc. Concentrez vous sur vos propres problèmes. Si votre programme est refusé par le script de vérification, ne paniquez pas. Réfléchissez à ce que pourrait bien être la cause du problème. N'oubliez pas que les résultats produits doivent être identiques à ceux attendus (modulo, dans certains cas, les espaces blancs), puisque la vérification des résultats se fait avec `diff` (ou `diff -w`).

Un autre point à surveiller: pour certains problèmes plus complexes, le temps d'exécution ne doit pas dépasser une certaine limite de temps. Il peut donc être important, dans certains cas, de choisir le bon algorithme, c'est-à-dire, avec la bonne complexité asymptotique. Il n'est pas nécessaire de produire le programme le plus court possible (ce n'est pas un concours de style de programmation) ou celui qui fonctionne le plus rapidement : le programme doit simplement satisfaire l'énoncé du problème, pas plus, pas moins, et ce sans dépasser la limite de temps prévue.