TP d'algorithmique avancée
TP 1 : complexité et temps d'exécution

Jean-Michel Dischler et Frédéric Vivien

Recherche simultanée du minimum et du maximum

1. Écrivez un programme qui implémente d'une part l'algorithme naïf de recherche simultanée du minimum et du maximum, et d'autre part l'algorithme optimal vu en TD (si besoin est, le corrigé du TD est disponible à l'URL : `http://icps.u-strasbg.fr/~vivien/Enseignement/Algo-2001-2002/index.html`).

   Comparez les temps d'exécution des deux algorithmes (par exemple en utilisant la fonction `clock`). Qu'observez-vous ? Qu'en concluez-vous ?
2. Récrivez le programme précédent en remplaçant les comparaisons au moyen des opérateurs $>$ et $<$ par des appels à une fonction `compare(a, b)` qui renvoie la valeur du test « `a` $>$ `b` ».

   Qu'observez-vous ? Qu'en concluez-vous ?

Calcul de x^n

1. Écrivez un programme qui implémente d'une part l'algorithme naïf de calcul de x^n et d'autre part la « méthode binaire » vue en cours (si besoin est, le cours est disponible à l'URL : `http://icps.u-strasbg.fr/~vivien/Enseignement/Algo-2001-2002/index.html`).

   Pour implémenter cet algorithme, vous avez besoin de pouvoir récupérer le i^{e} bit d'un entier. Pour ce faire vous pouvez utiliser les décallages : $n >> i$ est équivalent à une division par 2^i et amène donc le i^{e} bit en position de poids faible ; et l'opérateur « & » qui est le *et* logique bit à bit.

   Comparez les temps d'exécution des deux algorithmes (par exemple en utilisant la fonction `clock`). Qu'observez-vous ? Qu'en concluez-vous ?
2. Pour pouvoir comparer effectivement les deux algorithmes, à la question précédente, il vous a fallu utiliser des valeurs de la puissance, n, relativement élevées. De ce fait, les résultats des calculs étaient forcément faux et généraient des dépassement de capacités (*overflow*), à moins de n'essayer de ne calculer que des puissances de 0, 1 ou -1.

   Pour remédier à ce problème, récrivez votre programme en utilisant la librairie *gmp* de GNU qui permet de faire du calcul en précision arbitraire. Pour ce faire, vous avez uniquement besoin d'inclure le fichier de déclaration idoine (`gmp.h`), de lier votre application avec la librairie *ad-hoc* (`gmp`), et d'utiliser les fonctions et constructions appropriées, celles listées ci-dessous suffisant amplement :
   - `mpz_t q` : déclare l'entier `q` en précision arbitraire ;
   - `mpz_init(mpz_t q)` : effectue l'initialisation de l'entier `q`, cette initialisation est indispensable ;
   - `mpz_set_ui(mpz_t q, unsigned long x)` : affecte la valeur de `x` à `q` ;
   - `mpz_mul(mpz_t a, mpz_t b, mpz_t c)` : effectue la multiplication de `b` et de `c` et stocke le résultat dans `a`.
   - `mpz_out_str(FILE * stream, int base, mpz_t q)` : affiche sur le flot de sortie `stream` (typiquement `stderr` ou `stdout`) la valeur de l'entier `q` exprimée dans la base `base` qui doit être un entier compris entre 2 et 32.

   Comparez les temps d'exécution des deux algorithmes. Qu'observez-vous ? Qu'en concluez-vous ?