R5.04 - Qualité algorithmique

TP 2 - Mesure du temps d'exécution

2025-2026

Dans ce TP, nous allons réaliser une implémentation des listes chainées vues en TD. Les enjeux sont multiples :

— vous faire implémenter une structure de données très classique
— réaliser des premières mesures de performances à l'aide d'outils standard

Vous trouverez sur Célène un fichier d'en-tête (`.h`) ainsi que des ressources et rappels concernant les listes chainées.

Implémentation

Reprenez le travail réalisé en TD avec votre enseignant. Vous devez implémenter toutes les fonctions déclarées dans l'en-tête fourni.

Pour la compilation, vous réaliserez un Makefile afin d'automatiser la génération de l'exécutable.

Vous veillerez notamment à ne pas oublier les options suivantes :

```
-Wall -Wextra -Wconversion -g -O0
```

La compilation doit se terminer sans message d'avertissement.

Evaluation expérimental

Temps d'exécution

La méthode la plus courante pour mesurer le temps d'exécution d'un programme est de le "chronométrer". Cependant, plusieurs programmes sont en cours d'exécution en même temps sur votre système et se partagent le CPU. Il faut donc privilégier la mesure du **temps d'utilisation CPU** plutôt que le temps absolu.

La bibliothèque time.h

Pour cela, il existe en C la bibliothèque `time.h`, qui possède notamment la fonction `clock()`.

Cette fonction va renvoyer une approximation du temps CPU utilisé par le programme, en terme de tic d'horloge. Pour avoir une approximation en seconde, il est nécessaire de diviser la valeur obtenue par `CLOCKS_PER_SEC`. Voici un exemple d'utilisation :

```
#include <time.h>

clock_t debut = clock();
// instructions du programme
clock_t fin = clock();
double temps = (double)(fin - debut) / CLOCKS_PER_SEC;
```

Une autre fonction permet de mesurer le temps d'exécution de manière plus précise. Il s'agit de la fonction `clock_gettime()`. L'exemple suivant est partiellement tiré du site cppreference.

```
struct timespec ts_debut; // contient un champ tv_sec (seconde) et un champ tv_nsec (nan
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts_debut);
// instructions
struct timespec ts_fin;
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts_fin);
double temps_s = ts_fin.tv_sec- ts_debut.tv_sec ;
double temps_ns = ts_fin.tv_nsec - ts_debut.tv_nsec ;
printf("temps (seconde): %f\n",temps_s);
printf("temps (nanoseconde): %f\n",temps_ns);
printf("temps: %f\n",temps_s + temps_ns*1e-9);
```

1. Créer un fichier `main.c` qui inclura notamment votre fichier d'en-tête `liste.h`. Votre fonction `main()` devra mesurer le temps d'exécution des différentes opérations sur les listes.
    1. Proposez un protocole expérimental permettant une mesure pertinente du temps d'exécution de chaque opération
    2. Comparez les mesures effectuées par les deux méthodes précédentes (`clock()` et `clock_gettime()`).

L'outil callgrind de valgrind

Les exemples précédents permettent d'approximer le temps d'exécution d'un programme. Toutefois, cette mesure est par nature complètement dépendante du contexte d'exécution ! Le même programme ne s'exécutera pas à la même vitesse d'une machine à l'autre... Même

sur une seule machine, il n'aura probablement pas le même temps mesuré d'une exécution à l'autre !

Une mesure moins dépendante de l'environnement d'exécution consiste à compter le nombre d'instructions exécutées par votre processeur. Cette méthode se rapproche de l'évaluation théorique des algorithmes, où on cherche à dénombrer le nombre d'instructions élémentaires effectuées par un algorithme.

Pour cela, `valgrind` propose un outil nommé `calgrind` qui permet d'analyser votre programme C et qui détermine le nombre d'instructions CPU pour chaque instruction de votre programme.

Vous trouverez des détails d'utilisation ici et là.

Pour lancer le programme :

```
valgrind --tool=callgrind prog
```

`callgrind` va générer un rapport nommé `callgrind.out.pid` où `pid` correspond au pid de votre programme.

Pour pouvoir interpréter ce rapport, il faut l'annoter :

```
callgrind_annotate --auto=yes callgrind.out.pid
```

1. Lancez `callgrind` sur l'exécutable réalisé précédemment à partir de votre fichier `main.c`.
2. Relevez le nombre total d'instructions pour chaque opération. Comparez vos résultats avec ceux de vos voisins. Qui a l'algorithme le plus rapide ?

Optimisation du compilateur

Le compilateur `gcc` possède une option d'optimisation noté `-OX`, où `X` est un nombre entier entre 0 et 3. Sans rentrer dans les détails du rôle de ces différents niveaux d'optimisation, nous allons simplement en mesurer les conséquences sur le temps d'exécution.

1. Relancer vos évaluations en remplacant l'option `-O0` par `-O1`. Quelle différence constatez-vous ? et avec les options `-O2` et `-O3` ?

En bref, le compilateur va générer un code assembleur différent en fonction du niveau d'optimisation. Bien souvent, un programme plus rapide devient également plus long et donc prend plus de place en mémoire... La mesure de l'occupation mémoire sera abordé dans la prochaine et dernière partie de tp.