TP/TD #6 — Mesure de temps d'exécution & analyse de coût (avec solutions)

Le but de ce TP/TD est d'analyser (théoriquement) le coût d'un certain nombre d'algorithmes que vous avez (pour la plupart) déjà implémenté au cours des TPs des semaines précédentes. En parallèle de cette analyse théorique vous devrez aussi mesurer ces coûts expérimentalement, grâce à des mesures de temps d'exécution.

Outils de mesure de temps de calcul

La façon la plus simple de mesurer un temps de calcul est d'utiliser le programme *time* en ligne de commande UNIX. Celui-ci s'invoque comme `> time cmd` avec `cmd` n'importe quelle commande (demandant par exemple l'exécution d'un programme), et affiche le temps pris par l'exécution de celle-ci :

```
> time ./a.out
./a.out  0.17s user 0.00s system 99% cpu 0.177 total
```

Comme le montre l'exemple ci-dessus, le temps est décomposé en plusieurs catégories (*user*, *system*, *total*) ; dans notre cas et pour un programme faisant peu d'entrées/sorties (n'affichant rien ou peu de choses à l'écran, et ne sollicitant pas d'entrée de la part des utilisateurs ou utilisatrices), la catégorie *system* devrait être proche de zéro, et l'on pourra se contenter de consulter (assez indifféremment) la catégorie *user* ou *total*.
Un inconvénient de cette approche est qu'elle ne permet que de mesurer le temps *total* de l'exécution d'un programme : dans le cas où l'on souhaiterait mesurer le temps de plusieurs fonctions, il faut préparer plusieurs programmes utilisant chacun une unique fonction parmi celles-ci, et les exécuter et mesurer tout à tour.
Un autre inconvénient est que la précision de la mesure effectuée par *time* n'est pas très élevée ; pour mesurer le temps d'exécution d'une fonction « trop rapide », il est typiquement nécessaire d'exécuter la fonction un grand nombre de fois (en l'appelant dans une boucle) afin d'obtenir un temps d'exécution réel d'au moins quelques dixièmes de secondes, puis de moyenner. Dans ce cas, il faut cependant prendre garde à ne répéter *que* l'exécution de la fonction (et pas, par exemple, une initialisation coûteuse).

Méthodologie de mesures expérimentales

Le but des mesures expérimentales demandées dans ce sujet est d'estimer (ou confirmer) une tendance *asymptotique*. Pour cela il est nécessaire d'effectuer plusieurs mesures pour des valeurs croissantes du paramètre en lequel s'exprime le coût (généralement, la taille de l'entrée). Il « suffit » ensuite d'interpréter les résultats (éventuellement en s'aidant d'un graphique, mais ce n'est pas nécessaire). Par exemple :

- dans le cas d'une tendance linéaire (en $\Theta(n)$), doubler la valeur du paramètre double le temps d'exécution ;
- dans le cas d'une tendance quadratique (en $\Theta(n^2)$), doubler la valeur du paramètre multiplie le temps d'exécution par quatre ;
- plus généralement, dans le cas d'une tendance en $\Theta(n^k)$, doubler la valeur de n multiplie le temps d'exécution par 2^k ;
- dans le cas d'une tendance exponentielle, augmenter la valeur du paramètre *de un* multiplie le temps d'exécution par une certaine constante ;
- dans le cas d'une tendance logarithmique (par ex. en $\Theta(\log n)$), *multiplier* la valeur du paramètre par deux *augmente* le temps d'exécution d'une certaine constante ;
- dans le cas d'une tendance *quasi-linéaire* (par ex. en $\Theta(n \log n)$), multiplier n par un « grand » nombre multiplie le temps d'exécution par « un peu plus » que ce nombre.

On remarquera qu'une éventuelle difficulté d'une telle approche expérimentale est de bien choisir les valeurs d'entrées pour les fonctions évaluées : si l'on souhaite estimer un coût *dans le pire cas*, il faut s'assurer que celles-ci correspondent bien à un pire cas (ce qui peut être plus ou moins simple à garantir). Ainsi, effectuer des mesures expérimentales pertinentes et de bonne qualité nécessite souvent une bonne compréhension de la fonction mesurée (au même titre qu'une analyse de coût théorique).

CONSIGNE : les questions (ou parties) marquées d'une ⋆ sont à réaliser après *toutes* les autres.

Recherche linéaire

1. Montrez par une analyse de coût théorique *et* des mesures expérimentales que les fonctions (adaptées du TP#2) :

— `int max(size_t an, int a[an])`
— `bool search(size_t an, int a[an], int x)`

ont un coût au plus linéaire en an dans le pire cas, c'est à dire un coût qui est un $O(an)$.

Solution partielle : analyse théorique de `search`. On se donne par exemple :

```
bool search(size_t an, int a[an], int x)
{
    for (size_t i = 0; i < an; i++)
    {
        if (t[i] == x)
        {
            return true;
        }
    }
    return false;
}
```

Version longue : `search` contient une unique boucle `for` dont le nombre d'itérations est contrôlé par `i < an`. La variable `i` étant initialisée à `0` et étant incrémentée de 1 à chaque itération, le nombre d'itération est au plus an. Les instructions exécutées dans le corps de boucle sont toutes de coût constant, et le coût total de cette boucle est un $O(an)$ dans le pire cas (qui correspond au cas où le test effectué à l'intérieur de la boucle ne s'évalue jamais à `true`). L'unique instruction exécutée hors de la boucle étant aussi de coût constant, le coût total de la fonction est un $O(an)$.

Version courte : `search` contient une unique boucle `for` qui effectue au plus an itérations (évident). Les instructions exécutées dans le corps de boucle sont toutes de coût constant, et le coût total de cette boucle est un $O(an)$ dans le pire cas. L'unique instruction exécutée hors de la boucle étant aussi de coût constant, le coût total de la fonction est un $O(an)$.

Recherche logarithmique ⋆

2. Montrez par une analyse de coût théorique *et* des mesures expérimentales que la fonction (du TD#3) :

— `int bsearch(int n, int a[n], int e)`

a un coût au plus logarithmique en an dans le pire cas, c'est à dire un coût qui est un $O(\log n)$.

On rappelle `bsearch` :

```
1  int bsearch(int n, int a[n], int e) {
2      int b = 0;
3      int t = n - 1;
4
5      while (b < t) {
6          int m = b + (t - b)/2;
7          if (e == a[m]) {
8              return m;
9          }
10         if (e < a[m]) {
11             t = m - 1;
12         }
13         else {
14             b = m + 1;
15         }
16     }
17
18     if ((b > t) || (a[b] != e)) {
19         return -1;
```

```
20      }
21      return b;
22  }
```

Dans tout ce qui suit, on note :

— $a \div b$ le quotient de la division euclidienne de a par b, tous deux positifs. (Attention : notation peu usuelle, **à définir avant d'utiliser dans une copie de concours par exemple**.)
— $\lfloor x \rfloor$ la partie entière d'un réel x. (Notation au programme de maths.)
— $\{x\}$ la partie fractionnaire d'un réel x. (À rappeler.)

En particulier (pour $a, b, x \geq 0$), on a :

— $x \geq \lfloor x \rfloor$
— $0 \leq \{x\} < 1$
— $x = \lfloor x \rfloor + \{x\}$
— $a \div b = \lfloor a/b \rfloor$
— $a/b = a \div b + \{a/b\}$

La fonction `bsearch` contient une unique boucle `while` dont le corps est de coût constant, et quelques autres instructions de coût constant également. Son coût est donc donné par le nombre d'itérations de la boucle, et le pire cas correspond à celui où le test de la ligne 7 n'est jamais satisfait.
Soit b & b' et t et t' les valeurs des variables `b` et `t` en entrée et sortie d'une itération de boucle, on va montrer que $(t'-b')/(t-b) \leq C$ avec $C > 1$ une certaine constante. Ceci permettra de majorer le nombre d'itérations avant que l'on ait $|t-b| < 1$ par $\log_C(n-1)+1$, ce qui suffira pour conclure.
On considère les deux cas du test de la ligne 10 :

— S'il s'évalue à `true`, alors $b' = b$, $t' = b + (t-b) \div 2 - 1$, et :

$$\begin{aligned} t' - b' &= (t-b) \div 2 - 1 \\ &< (t-b)/2 \qquad \text{car} \quad (t-b) \div 2 \leq (t-b)/2 \end{aligned}$$

— S'il s'évalue à `false`, alors $b' = b + (t-b) \div 2 + 1$, $t' = t$, et :

$$\begin{aligned} t' - b' &= t - b - (t-b) \div 2 - 1 \\ &= (t-b) - (t-b)/2 + \{(t-b)/2\} - 1 \\ &= (t-b)/2 + \{(t-b)/2\} - 1 \\ &< (t-b)/2 \qquad \text{car} \quad \{(t-b)/2\} < 1 \end{aligned}$$

Dans les deux cas $(t'-b')/(t-b) < 2$, ce qui permet de conclure.

Tri linéaire

3. Montrez par une analyse de coût théorique *et* des mesures expérimentales que la fonction (du TP#4) :

— `void sag_sort(size_t an, int a[an])`

a un coût au plus linéaire en an dans le pire cas. (Si ce n'est pas le cas, cela veut dire que vous pouvez améliorer votre fonction ! Faites le.)

Pour `sag_sort` on propose :

```
void sag_sort(size_t an, int a[an])
{
    unsigned cntz = 0;

    for (size_t i = 0; i < an; i++)
    {
        if (a[i] == 0)
        {
            cntz += 1;
        }
```

```
    }

    size_t i = 0;
    for (; i < cntz; i++)
    {
        a[i] = 0;
    }
    for (; i < an; i++)
    {
        a[i] = 1;
    }

    return;
}
```

Analyse de coût (version courte) : `sag_sort` contient trois boucles `for` successives, dont les corps sont tous de coût constant, et uniquement une instruction de coût constant en dehors de ces boucles. Son coût est donc donné par la somme des coûts des boucles. La première boucle effectue an itérations (évident), et à sa sortie l'on a $cntz \leq an$. Il s'ensuit que la seconde boucle effectue au plus an itérations, de même que la troisième (évident). Le coût total est donc un $O(an)$.

Tris quadratiques

4. Montrez par une analyse de coût théorique *et* des mesures expérimentales que les fonctions (du TP#4) :

- `void select_sort(size_t an, int a[an])`
- `void bubble_sort(size_t an, int a[an])`

ont un coût au plus quadratique en an dans le pire cas, c'est à dire un coût qui est un $O(an^2)$

Pour `select_sort` on propose :

```
1  void select_sort(size_t an, int a[an])
2  {
3      assert(an > 0); // required by law
4      for (size_t i = 0; i < an - 1; i++)
5      {
6          int min = a[i];
7          int minix = i;
8          for (size_t j = i + 1; j < an; j++)
9          {
10             if (a[j] < min)
11             {
12                 min = a[j];
13                 minix = j;
14             }
15         }
16         int t = a[i];
17         a[i] = min;
18         a[minix] = t;
19     }
20
21     return;
22 }
```

Analyse de coût (version très courte, qui n'effectue pas une analyse très précise mais est suffisante pour répondre à la question) : On s'intéresse d'abord à la boucle interne des lignes 8 à 15 : par le typage de `j` et la condition d'arrêt, celle-ci effectue au plus an itérations, et a un corps de coût constant ; son coût est donc un $O(an)$. Pour la même raison et par l'`assert` de la ligne 3, la boucle externe effectue au plus $an - 1 \geq 0$ itérations, dont le coût est une constante plus une exécution de la boucle interne. Son coût (et celui de la fonction toute entière) est donc un $O(an^2)$.

Exponentiation (pas) rapide ⋆

5. Montrez par une analyse de coût théorique *et* des mesures expérimentales que les fonctions (adaptées du DM#1 en fixant n à 4 294 967 291) :

— `uint64_t slow_expmod(uint64_t x, uint64_t e)`
— `uint64_t fast_expmod(uint64_t x, uint64_t e)`

ont respectivement un coût au plus exponentiel et linéaire en la taille « effective » de e (c'est à dire, le nombre *minimal* de chiffres nécessaires à son écriture en base deux). Plus précisément, soit n cette taille, montrez que la première fonction a un coût qui est un $O(2^n)$, et la seconde un coût qui est un $O(n)$.

On propose les fonctions :

```
uint64_t slow_expmod(uint64_t x, uint64_t e)
{
    uint64_t mod = 4294967291;
    uint64_t r = 1;
    x = x % mod;

    for ( ; e > 0; e--)
    {
        r = (r * x) % mod;
    }

    return r;
}

uint64_t fast_expmod(uint64_t x, uint64_t e)
{
    uint64_t mod = 4294967291;
    uint64_t r = 1;
    x = x % mod;

    for ( ; e > 0; e /= 2)
    {
        if (e % 2 == 1)
        {
            r = (r * x) % mod;
        }
        x = (x * x) % x;
    }

    return r;
}
```

Analyse de coût de `slow_expmod` : similaire aux exemple ci-dessus, il faut juste s'assurer que la condition de sortie (portant sur une variable de type non signé) et la décrémentation de celle-ci ne cause pas de boucle infinie par *underflow*. Cela étant dit, le coût est un $O(e)$ (évident).

Analyse de coût de `fast_expmod` : le corps de l'unique boucle `for` est de coût constant, et le coût est donc donné par le nombre d'itération. On « observe » alors que c'est le nombre de fois que l'on peut diviser `e` par 2 euclidiennement avant d'obtenir un quotient non nul, ce qui est un $O(\log e)$ (si l'on voulait être plus rigoureux, on pourrait prouver cela en utilisant un invariant similaire à celui du TD #4). On conclut en observant que la taille effective de e est elle aussi un $O(\log e)$.

Élément le plus présent

6. Analysez théoriquement *et* par des mesures expérimentales le coût de votre fonction (du TP#5):

— `int maj(size_t n, int a[n])`

Crible d'Erathostène

Le *crible d'Erathostène* est un algorithme antique de recherche de nombres premiers. Son principe est le suivant : pour trouver *tous* les nombres premiers inférieurs à un certain entier B, on initialise un tableau de B booléens à *vrai* (sauf pour les indices inférieurs stricts à deux, qui sont initialisés à *faux*), et pour les entiers i croissants de deux à B (ou même $\lceil \sqrt{B} \rceil$, mais cela ne change pas grand chose au coût) : si la case d'indice i du tableau est à *faux*, alors i n'est pas premier et l'on passe à l'entier suivant ; sinon i est premier, et l'on modifie le tableau pour prendre en compte le fait que ses multiples ne sont pas premiers, avant de passer à l'entier suivant. Une fois que tous les entiers ont été parcourus, les indices des cases à *vrai* du tableau sont exactement les nombres premiers inférieurs à B.

7. Écrivez une fonction C de signature :
`void eratho_sieve(size_t b, bool primes[b])`
qui implémente l'algorithme décrit ci-dessus : elle prend en entrée un tableau de booléens (non initialisé) de longueur b, et le modifie de sorte qu'après son exécution *primes[i]* vaut *vrai* ssi. i est premier.

On propose :

```
1  void eratho_sieve(size_t b, bool primes[b])
2  {
3      assert(b > 0);
4      primes[0] = false;
5      if (b == 1)
6      {
7          return;
8      }
9      primes[1] = false;
10     for (size_t i = 2; i < b; i++)
11     {
12         primes[i] = true;
13     }
14
15     for (size_t i = 2; i < b; i++)
16     {
17         if (primes[i])
18         {
19             for (size_t j = 2 * i; j < b; j += i)
20             {
21                 primes[j] = false;
22             }
23         }
24     }
25
26     return;
27 }
```

8. Testez, en utilisant au moins une fonction de test.

9. Analysez sont coût théoriquement *et* par des mesures expérimentales, en fonction de la *valeur* de b. Votre analyse théorique doit vous permettre de montrer que ce coût est « plus faible que quadratique » (autrement dit, vous devez pouvoir l'exprimer sous la forme $O(e)$ avec e tel que la fonction $n \mapsto n^2$ n'est **pas** un $O(e)$). Si ce n'est pas le cas, cela veut dire que vous pouvez améliorer votre fonction et/ou votre analyse.
Pour l'analyse théorique, on pourra utiliser les deux approximations suivantes (admises) :

— $H_n := \sum_{i=1}^{n} 1/i = O(\log n)$
— $\sum_{i \in \mathcal{P}_n} 1/i = O(\log \log n)$, où $\mathcal{P}_n$ désigne l'ensemble des nombres premiers inférieurs à n (Théorème de Mertens)

Les lignes 3 à 13 ont collectivement un coût en $O(b)$, et l'on peut se concentrer sur la boucle L_{15} des lignes 15 à 24. Celle-ci contient une boucle interne L_{19} (lignes 19 – 21) que l'on suppose dans un premier temps exécutée à chaque itération de la boucle englobante L_{15}. Le coût d'une itération de L_{19} étant constant, il suffit de compter son

nombre d'itérations, que l'on peut aisément déterminer comme étant un $O(b \div i)$. Ainsi, le coût total (à constantes près) de L_{15} est donné par :

$$\sum_{i=2}^{b} \frac{b}{i} = b \sum_{i=2}^{b} \frac{1}{i} \leq bH_b = O(b \log b)$$

On peut ensuite légèrement améliorer cette analyse en constatant (pour être rigoureux, il faudrait bien évidemment le prouver (par exemple en utilisant un invariant adapté)) que L_{19} n'est exécutée que si i est premier, et donc que le coût de L_{15} est donné par :

$$\sum_{i \in \mathcal{P}_b} \frac{b}{i} = b \sum_{i \in \mathcal{P}_b} \frac{1}{i} = O(b \log\log b)$$

Remarque : on pourrait légèrement améliorer l'algorithme proposé ci-dessus en prenant pour condition d'arrêt de L_{15} $i \leq \sqrt{b}$, mais cela n'améliorerait pas le coût asymptotique *à constantes près* : $O(b \log\log \sqrt{b}) = O(b \log\log b)$.

10. Quels avantages & inconvénients éventuels peut avoir le crible d'Erathostène par rapport aux algorithmes (plus ou moins naïfs) étudiés dans le DM#1 ? Notamment si l'on cherche à déterminer si un unique nombre est premier ? Si l'on cherche à trouver un grand nombre de nombres premiers ?

Un inconvénient de taille est que le coût en espace du crible d'Erathostène est (essentiellement) égal à son coût en temps : en pratique, il saturera rapidement la mémoire. Un avantage est cependant qu'il permet de tester si un nombre est premier en coût *amorti* (quasiment) constant : cela prend $O(n \log\log n)$ opérations pour tester la primalité de n nombres (on peut difficilement espérer mieux). Malheureusement ceci reste relativement peu utile : les problèmes pratiques nécessitant des tests de primalité ont souvent besoin de seulement tester la primalité de *quelques* nombres pouvant être très grands (par exemple plus grands que 2^{2000}).

Fusion de tableaux

Le *tri fusion* est un algorithme de tri de coût pire cas quasi-linéaire en le nombre d'éléments à trier : il permet de trier un tableau de n éléments en temps $O(n \log n)$, ce qui est nettement mieux que le tri par sélection ou le tri par bulle, par exemple. Une opération essentielle du tri fusion consiste à *fusionner* deux tableaux triés : soit t_1, t_2 de tels tableaux, la *fusion* de t_1 et t_2 consiste à calculer un nouveau tableau $t_{1,2}$ contenant les mêmes éléments que t_1 et t_2 (avec multiplicité), et qui est lui-même trié.

11. Écrivez une fonction C :

```
void merge(size_t an, int a[an],
           size_t bn, int b[bn],
           int ab[an + bn])
```

qui prend en entrée deux tableaux triés `a` et `b` de longueur respectivement `an` et `bn`, ainsi qu'un tableau `ab` de longueur `an + bn`, et qui écrase le contenu de `ab` avec celui de la fusion triée de `a` et `b`.

On propose :

```
1  void merge(size_t an, int a[an],
2             size_t bn, int b[bn],
3             int ab[an+bn])
4  {
5      size_t ia = 0;
6      size_t ib = 0;
7      size_t iab = 0;
8
9      while (ia < an && ib < bn)
10     {
11         if (a[ia] < b[ib])
12         {
13             ab[iab] = a[ia];
14             ia = ia + 1;
15         }
16         else // b[ib] >= a[ia]
```

```
17          {
18              ab[iab] = b[ib];
19              ib = ib + 1;
20          }
21          iab = iab + 1;
22      }
23      while (ia < an)
24      {
25          ab[iab] = a[ia];
26          ia = ia + 1;
27          iab = iab + 1;
28      }
29      while (ib < bn)
30      {
31          ab[iab] = b[ib];
32          ib = ib + 1;
33          iab = iab + 1;
34      }
35
36      return;
37  }
```

12. Testez, en utilisant au moins une fonction de test.

13. Analysez le coût de `merge` théoriquement et expérimentalement, et exprimez le en fonction de paramètres pertinents.

> Analyse rapide :
> Les trois boucles `while` ont un nombre d'itération respectivement inférieur à $an+bn$, an, bn (« évident » de par leurs conditions d'arrêt et le fait que chaque itération de la première boucle incrémente `ia` ou `ib`, et chaque itération de la seconde (troisième) incrémente `ia` (`ib`)). Tous les corps de boucle étant de coût constant, il s'ensuit que le coût total de `merge` est un $O(an+bn)+O(an)+O(bn)$, soit un $O(an+bn)$: il est linéaire en la taille de ses entrées.

Sous-tableau consécutif de valeur maximale ⋆⋆

Soit t un tableau de n entiers *signés* (pouvant prendre des valeurs négatives, sinon le problème n'est pas très intéressant), on souhaite trouver le sous-tableau d'indices consécutifs de t maximisant la somme des éléments de t de ces indices. Autrement dit, on cherche à trouver deux entiers i, j avec $0 \leq i \leq j < n$ qui maximisent $\sum_{k=i}^{j} t[k]$.

14. Proposez un algorithme qui résout ce problème et dont le coût est un $O(n^2)$.

> Il suffit de calculer les valeurs de tous les $O(n^2)$ sous-tableaux consécutifs possibles, en calculant les sommes de façon incrémentale.

15. Proposez un algorithme (éventuellement différent du précédent) qui résout ce problème et dont le coût est un $O(n)$.
INDICES :

- Comment peut-on caractériser la valeur maximale de la somme des éléments de tous les sous-tableaux consécutifs se terminant en j en fonction de la valeur maximale de la somme des éléments de tous les sous-tableaux consécutifs se terminant en $j-1$?
- Bien que cela ne soit pas vraiment nécessaire, il peut être plus simple d'exprimer cet algorithme en utilisant un tableau auxiliaire supplémentaire.

16. Implémentez votre algorithme de la question précédente en C. Vous êtes libres de choisir la signature de la fonction (vous pouvez par exemple inclure un paramètre tableau servant de stockage pour un tableau auxiliaire), et notamment de la façon dont elle communique son résultat (par exemple en renvoyant une somme maximale, ou les indices i et j, etc.).

On propose :

```
struct array
{
    int *start;
    size_t len;
};

struct array maxsuba(size_t n, int a[n])
{
    struct array maxsub = {.start = a, .len = 0};
    int maxval = 0;
    struct array sub = maxsub;
    int val = 0;

    for (size_t i = 0; i < n; i++)
    {
        if (a[i] >= 0)
        {
            if (val <= 0)
            {
                sub.start = &a[i];
                sub.len = 1;
                val = a[i];
            }
            else
            {
                sub.len = sub.len + 1;
                val = val + a[i];
            }

            if (val > maxval)
            {
                maxval = val;
                maxsub = sub;
            }
        }
        else
        {
            int nval = val + a[i];
            if (nval > 0)
            {
                val = nval;
                sub.len = sub.len + 1;
            }
            else
            {
                val = 0;
                sub.start = &a[i];
                sub.len = 0;
            }
        }
    }

    return maxsub;
}
```

Ce n'est pas le plus simple, mais ça permet d'être en coût en espace (supplémentaire) constant et de travailler un peu les pointeurs.

17. Testez, en utilisant au moins une fonction de test.