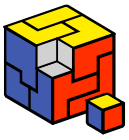
Prologin 2010



Concours National d'Informatique

Questionnaire de sélection

Correction des questions d'algorithmique

Raphael Marinier Thomas Deniau*

18 janvier 2010

1 Nucléotide

1.1 Énoncé

On vous donne en entrée une séquence d'ADN de longueur N. Écrivez une fonction qui renvoie le nucléotide (la lettre) le plus présent. Si c'est le cas de plusieurs, renvoyez celui qui vient en premier dans l'ordre alphabétique.

1.2 Solutions proposées par les candidats

Nous avons observé des dizaines d'implémentations différentes de cet exercice simple, des plus simples aux très complexes. En général, sachez que si vous devez écrire plus de dix lignes de code pour résoudre le premier exercice du questionnaire, vous vous trompez…

L'idée était bien sûr de maintenir dans un tableau des compteurs correspondant aux quatre lettres, puis de le parcourir pour trouver son maximum. Dans les mauvaises solutions, l'on a par exemple trouvé :

- une fois les lettres comptées, le tri du tableau des comptes afin de trouver son maximum : le maximum d'un tableau se trouve en temps linéaire (il suffit de parcourir tout le tableau en comparant chaque élément au maximum courant), alors qu'un tri a une complexité minimum $O(n \log n)$ sauf dans certains cas particuliers ;
- le tri de la chaîne elle-même pour compter les lettres plus facilement !
- le parcours quatre fois de la chaîne pour compter chaque lettre indépendamment ;
- un parcours du maximum avec plusieurs boucles imbriquées.

Nous avons également rencontré beaucoup de copies utilisant quatre variables pour compter les occurrences de chaque nucléotide indépendamment : c'était correct, mais allongeait inutilement le code. Un tableau s'y prêtait mieux, surtout pour le calcul du maximum ensuite.

Rappelons ici l'utilité de l'utilisation des bibliothèques standard. Elles vous permettent de gagner quelques lignes, et donc de gagner en rapidité d'écriture et de rencontrer moins de bogues.

Cependant, dès cet exercice, on a trouvé des personnes en abusant. Ainsi, lorsqu'il s'agit de compter les différents caractères d'une chaîne, l'utilisation de la fonction **substr_count** de PHP est

*Pour l'équipe des correcteurs de Prologin 2010 : Pierre Bourdon, Laurent Le Brun, Thomas Deniau, Matthieu Kermagoret, Thomas Lecomte, Raphaël Marinier, Monirath Pontiac, Thomas Refis, Alexis Rolland, etc.

Listing 1 : Solution de l'exercice 1 en C++

```
char MostFrequentNucleotid(const std::string& s) {
  int frequencies[256] = {0};  // met tout le tableau à 0
  for (int i = 0 ; i < int(s.size()) ; ++i) {
    ++frequencies[s[i]];
  }
  // Renvoie la première (dans l'ordre alphabétique)
  // lettre la plus fréquente.
  return std::max_element(frequencies, frequencies + 256) - frequencies;
}
```

- exagérée : c'est tout l'intérêt de l'exercice ;
- inadaptée : elle compte les occurrences d'une sous-chaîne, non d'un caractère, et est donc bien trop lente pour la tâche demandée.

Utilisez vos bibliothèques standard à condition qu'elles soient adaptées !

1.3 Solution

Une bonne solution, au lieu de maintenir quatre compteurs indépendamment, est d'indexer un tableau par le code ASCII des caractères rencontrés. En C++, cela donne le code dans le listing 1.

Notez que la fonction `std::max_element` de la bibliothèque standard du C++ (dans la section <algorithm>) renvoie un pointeur vers l'élément maximal du tableau. Lui soustraire un pointeur vers le début du tableau (`frequencies`) permet donc d'obtenir le code ASCII de la lettre la plus fréquente.

Pour éviter, cependant, de parcourir tout un gros tableau pour trouver le maximum alors que l'on sait que l'on se restreint à certaines lettres, l'on pouvait mettre en place deux indirections : un tableau qui associe à chaque lettre importante (A, C, G, T) un entier entre 0 et 3 et un faisant l'inverse.

L'on peut alors avoir la même technique que dans la solution donnée avec simplement un maximum sur 4 éléments, ce qui est bien plus rapide.

2 Les acides aminés

2.1 Énoncé

Lors de la traduction (simplifiée !) d'une séquence d'ADN en suite d'acides aminés, chaque groupement de trois nucléotides de la séquence d'ADN est transformé en acide aminé. Écrivez une fonction qui, étant donné une séquence d'ADN de longueur N et la table de traduction de taille M, renvoie la suite d'acides aminés correspondante. On assure que tout le brin d'ADN pourra être traduit, que N est multiple de 3, et que la table de traduction est correcte.

2.2 Solution

Pour ce problème, la difficulté était d'utiliser une structure de données adaptée pour associer efficacement un acide aminé (une chaîne de caractères) à une suite de trois nucléotides.

Listing 2 : Solution de l'exercice 2 en Python

```
def traduction(sequence, table):
    # Découpe la sequence d'adn en une suite de sous-séquences de
    # trois nucléotides
    split_sequence = [sequence[i:i+3] for i in
                      range(0, len(sequence), 3)]
    # Utilise la table pour associer à chacun de ces groupements de
    # trois nucléotides l'acide aminé correspondant
    return " ".join([table[three_nucleotids]
                     for three_nucleotids in split_sequence])
```

Une première possibilité est de construire un tableau à trois dimensions contenant les acides aminés et indexé par le code ASCII de chacun des trois caractères. Une autre est d'utiliser une table associative, c'est à dire, en simplifiant, un « tableau indexé par des objets arbitraires ». On peut notamment utiliser une table associative pour associer une chaîne de caractères (acide aminé) à une autre chaîne de caractères (suite de trois nucléotides).

Notez que la plupart des langages proposent au moins une implémentation de cette structure de données dans leur bibliothèque standard. Les deux implémentations les plus répandues des tables associatives sont :

- Un arbre de recherche équilibré qui offre un coût en $O(\log(N))$ comparaisons d'éléments (où N est le nombre d'éléments dans la structure) pour toutes les opérations communes (ajout / recherche / modification / suppression d'un élément) ;
- Une table de hachage qui offre un coût en moyenne de $O(1)$ comparaisons d'éléments pour les opérations courantes.

Dans les langages que vous utilisez, nous rencontrons ainsi :

- En C++, std::map et `std::tr1::unordered_map` ;
- en Python, le type dict ;
- en Java, Hashtable et `HashMap` ;
- en OCaml, le module `Hashtbl`.

Nous vous conseillons de vous documenter sur l'utilisation de ces structures de données dans votre langage favori !

Un programme possible pour résoudre le problème en Python est donné dans le listing 2.

Encore une fois, on insiste sur l'utilité de maîtriser la bibliothèque standard et les idiomes du langage que vous utilisez pour pouvoir produire un code clair et concis !

L'on pouvait cependant faire encore mieux, à l'aide d'une idée qui peut également servir pour les autres exercices. En effet, nos séquences de nucléotides sont en nombre fini et assez restreint (ici $4^3 = 64$). On peut donc utiliser des entiers pour les repérer. Si l'on associe à chaque nucléotide un entier (tableau 1 page suivante), on peut associer le triplet de nucléotides (qui devient un triplet d'entiers compris entre 0 et 3) (a, b, c) au nombre $16a + 4b + c$. La chaîne « AGT » devient alors $16 \times 0 + 4 \times 2 + 3 = 11$. En écrivant 11 en base 4, l'on retrouve $\overline{023}$, soit « AGT ».

Chaque recherche de l'acide aminé correspondant à un codon devient alors un simple accès à un tableau indexé, comme d'habitude, par des entiers.

Nucléotide	Entier
A	0
C	1
G	2
T	3

Tab. 1 : Association d'un entier à chaque nucléotide

2.3 Solutions proposées par les candidats

Nous n'avons ici pas trouvé beaucoup de variations. La plupart des candidats ont hélas utilisé un algorithme très naïf, où pour traduire chaque codon, ils lisaient toute la table dans l'ordre jusqu'à trouver le bon. C'était une solution lente. Nous avons de plus rencontré dans ce cas quelques horreurs en C par ceux qui ne savaient pas utiliser les fonctions manipulant les chaînes…

La plupart des autres ont rendu un algorithme assez correct, utilisant des arbres de recherche ou des tables de hachage. La plupart des candidats codant en Python ont proposé une telle solution, probablement parce que la structure de données correspondante est au cœur du langage. Ce n'était cependant pas plus compliqué en C++ avec `std::map`.

Peu ont trouvé la solution utilisant des entiers décrite plus haut (qui était la solution attendue pour avoir le maximum de points). Nous avons également accordé le maximum à la solution utilisant un tableau à trois dimensions car elle était strictement équivalente à celle utilisant des entiers (en fait, un tableau à 64 cases indexées comme expliqué ci-dessus aura probablement exactement le même agencement mémoire qu'un tableau à trois dimensions), bien qu'elle soit moins généralisable.

Certains enfin n'ont pas compris qu'il fallait prendre la table de transcription en entrée, sont allés la chercher dans un livre de biologie, et l'ont codée en dur, souvent à l'aide d'une cascade de **if**s. L'effort est louable, mais il s'agit d'algorithmique, pas de code au kilomètre ou de biologie… Il s'agissait de tester votre connaissance des structures de données adaptées !

3 Sous-séquence

3.1 Énoncé

On cherche à analyser les fréquences d'apparition des sous-séquences d'une séquence d'ADN de longueur N donnée en entrée. Ecrivez une fonction qui renvoie la sous-séquence contiguë de longueur L de la chaîne d'ADN la plus fréquente. Dans le cas où plusieurs sous-séquences de longueur L apparaissent un même nombre de fois, affichez celle qui vient en premier dans l'ordre alphabétique.

Sur le serveur d'entraînement, nous avions fixé des contraintes sur les paramètres d'entrée : $1 \leq N \leq 200000$ et $1 \leq L \leq 15$, et la limite de temps à 40 millisecondes.

3.2 Solution

Bien que l'exercice soit d'apparence simple, il soulève beaucoup de questions non triviales. Mettre au point une solution correcte n'était pas difficile après avoir fait les deux premiers exercices. La difficulté résidait dans la limite de temps du serveur d'entraînement pour ce problème qui invitait le candidat à chercher une solution plus rapide.

Listing 3 : Première solution de l'exercice 3 en C++

```
std::string MostFrequentSubsequence(const std::string& adn_sequence, int L) {
  std::map<string, int> freqs;
  for (int i = L-1 ; i < adn_sequence.size() ; ++i)
    ++freqs[adn_sequence.substr(i - L + 1, L)];
  int max_freq = 0;
  std::string* best_subsequence = NULL;
  for (std::map<string,int>::const_iterator it = freqs.begin() ;
       it != freqs.end(); ++it) {
    if (it->second > max_freq) {
      max_freq = it->second;
      best_subsequence = &it->first;
    }
  }
  return *best_subsequence;
}
```

3.2.1 Une première solution assez rapide

Une première solution évidente au vu de l'exercice précédent est d'utiliser une table associative indexée par les chaînes de longueur L extraites de la séquence d'ADN, et comptant pour chacune de ces chaînes le nombre de fois qu'elles apparaissent. Un exemple d'une telle solution est donné en C++ dans le listing 3.

Commençons par calculer la complexité en temps de cette solution. Déjà, on extrait (par l'appel à string::substr) chaque sous-chaîne de longueur L de la séquence d'ADN donnée en entrée. Cela coûte $O(NL)$ opérations.

Ensuite, on incrémente le compteur associé à chacune de ces sous-chaînes (avec ++freq[sous_chaine]). Chaque recherche dans la table associative coûte au plus $O(\log(N))$ *opérations de comparaisons*, puisque la table associative contient au plus $N - L + 1$ éléments. Mais chaque opération de comparaison coûte au plus $O(L)$ opérations, puisque l'on compare des chaînes de taille L (d'autant plus que les chances sont élevées pour que beaucoup de chaînes soient quasiment identiques) ! Chaque recherche et incrémentation d'un élément dans la table associative coûte donc au total $O(\log(N)L)$ opérations.

On fait de telles recherches $O(N)$ fois ; le coût total de la première boucle est donc de $O(\log(N)NL)$ opérations. Enfin, la deuxième boucle consiste à extraire de la table associative la chaîne ayant le plus d'occurrences, ce qui coûte $O(N)$.

Le coût final est donc de $O(\log(N)NL)$. On peut le réduire à $O(NL)$ en utilisant une table de hachage au lieu d'une `map` du C++.

Voyons la durée d'exécution en pratique[1] du code donné en listing 3. Cela donne :

```
$ time ./a.out < test11.in
real 0m0.472s
user 0m0.448s
sys 0m0.024s
```

[1]Notre machine de test est un Intel Core Duo cadencé à 2 GHz, et les entrées sont lues avec `scanf`. Les codes sont compilés avec `g++` sous Linux avec toutes les optimisations (`-O3`). Les temps sont toujours donnés pour le test le plus lent parmi nos fichiers de tests, avec une séquence d'ADN de longueur 200 000.

Nucléotide	Entier	Représentation mémoire
A	0	00
C	1	01
G	2	10
T	3	11

Tab. 2 : Représentation de nucléotides en mémoire

et l'on s'aperçoit que c'est beaucoup trop lent pour être accepté sur le serveur d'entraînement, puisque la limite y est fixée à 40 millisecondes !

Essayons de remplacer `std::map<string, int>` par `std::unordered_map<string, int>`, plus rapide. On obtient :

```
$ time ./a.out < test11.in
real 0m0.271s
user 0m0.252s
sys 0m0.008s
```

ce qui n'est toujours pas suffisant pour passer sur le serveur d'entraînement.

C'est donc qu'il existe un algorithme plus rapide pour ce problème.

3.2.2 La solution attendue

Nous allons dans cette partie expliquer comment se débarrasser du facteur L dans la complexité de notre solution. L'idée est d'utiliser la correspondance donnée précédemment en tableau 1 page 4, mais aussi d'utiliser des opérations de manipulations de bits pour éviter les calculs inutiles. Le tableau 2 est une version modifiée du précédent donnant les représentations mémoire sur deux bits des entiers correspondant à chaque nucléotide.

L'idée naturelle suivante est de représenter une chaîne d'ADN de longueur L par la concaténation de notre nouvelle représentation en mémoire des lettres qui la composent. Par exemple, la chaîne « AAATGCGAC » se représente par $\overline{000001110011000001}$. Notez que vous obtenez la même représentation par le calcul avec les puissances de quatre expliqué lors de la correction de l'exercice précédent.

L'énoncé précise que L est plus petit que 15 ; chaque chaîne rencontrée peut donc se représenter comme une suite de 30 bits. Miracle, cela tient dans le type entier de la plupart des langages de programmation[2]. Nous allons donc réécrire tout l'algorithme en utilisant cette représentation, ce qui va nous permettre de gagner le facteur L de complexité annoncé, puisque nous aurons transformé la comparaison de chaînes de taille L en simple comparaison d'entiers (ce qui se fait en une seule opération machine).

On commence par réécrire la création du tableau de fréquences. Nous utilisons comme dans l'exercice 1 un tableau tr associant à chaque nucléotide l'entier correspondant. Il est important de respecter les valeurs données précédemment ; de cette manière, l'ordre est compatible avec l'ordre alphabétique, et l'on pourra plus facilement retourner la chaîne la plus petite selon l'ordre lexicographique.

Le code intéressant de cette partie est donné dans le listing 4 page suivante (en C++). La manipulation de bits demande probablement une explication ; à chaque nouvelle lettre, on

[2]Ces entiers font en général 32 bits. Attention à l'OCaml où ils font par défaut 31 bits, et au Pascal où le type standard des entiers fait parfois 16 bits !

Listing 4 : Deuxième solution pour l'exercice 3 en C++ - Première partie

```
1  typedef unsigned int uint;
2  std::string MostFrequentSubsequence(const std::string& adn_sequence, int L) {
3    const int N = adn_sequence.size();
4
5    // Associe à chaque chaîne de longueur L, representée sous la
6    // forme d'un entier, son nombre d'occurrences.
7    std::tr1::unordered_map<uint, int> freqs(N);
8
9    uint mask = (1<<(2*L)) - 1;
10
11   // [...] Création du tableau tr [...]
12
13   uint cur = 0;
14   for (int i = 0; i < N; ++i) {
15     cur <<= 2;
16     cur |= tr[adn_sequence[i]];
17     cur &= mask;
18     if (i >= L - 1) {
19       ++freqs[cur];
20     }
21   }
22   [...]
23 }
```

décale le nombre courant de deux bits vers la gauche pour « faire de la place » pour le nouveau nucléotide (ligne 15). On l'ajoute (ligne 16) puis on tronque le résultat à $2L$ bits à l'aide de la variable mask (ligne 17), ce qui a pour effet d'effacer la première lettre de la sous-chaîne. Nous avons donc enlevé la première lettre et ajouté la lettre suivante à l'autre bout : on a bien décalé la sous-chaîne, et ce en trois opérations très simples pour la machine.

Montrons un exemple avec $L = 3$, la séquence d'ADN « TGCGAC », et $i = 3$ dans la boucle principale de notre solution. `cur` vaut déjà $\overline{111001}$[3] ce qui représente la sous-chaîne « TGC ». La boucle modifie la variable `cur` pour représenter la sous-chaîne « GCG ». On commence par décaler `cur` de deux bits vers la gauche, ce qui donne $\overline{11100100}$. Ensuite, on utilise un OU-inclusif bit à bit (operation | en C++) pour ajouter à droite la traduction de la nouvelle lettre 'G' représentée par $\overline{10}$. `cur` devient $\overline{11100110}$. Enfin, la variable `mask` vaut $\overline{111111}$ et permet d'éliminer le $\overline{01}$ à gauche de la variable `cur` à l'aide de l'opérateur ET bit-à-bit (& en C++). `cur` vaut enfin $\overline{100110}$, ce qui est la représentation binaire de « GCG » !

La recherche du maximum se fait ensuite assez simplement (listing 5 page suivante).

La complexité de cette solution est dominée par la boucle principale, qui ne coûte maintenant que $O(N)$ opérations. On s'est affranchi de la comparaison de chaînes de caractères de longueur L, qui nous coûtait auparavant un facteur L !

Voyons ce que cela donne en pratique :

```
$ time ./a.out < test11.in
real 0m0.086s
user 0m0.084s
sys 0m0.004s
```

[3] Tous les entiers représentés ici sont implicitement complétés par des '0' à gauche.

Listing 5 : Deuxième solution pour l'exercice 3 en C++ - Deuxième partie

```
std::string MostFrequentSubsequence(const std::string& adn_sequence, int L) {
  [...]
  int max_freq = 0;
  uint best_subsequence;
  // Récuperation de la sous-chaîne la plus fréquente, représentée
  // sous la forme d'un entier.
  for (std::tr1::unordered_map<uint,int>::const_iterator
       it = freqs.begin(); it != freqs.end(); ++it) {
    if (it->second > max_freq ||
        it->second == max_freq && best_subsequence > it->first) {
      // Nous avons besoin de verifier l'ordre lexicographique, car
      // std::tr1::unordered_map n'est pas ordonnée comme son nom
      // l'indique !
      max_freq = it->second;
      best_subsequence = it->first;
    }
  }
  // Traduction inverse entier -> chaîne de caractères de la
  // sous-chaîne la plus fréquente.
  char back_tr[4];
  for (int x = 0 ; x < sizeof(tr) / sizeof(tr[0]) ; ++x) {
    if (tr[x] >= 0) back_tr[tr[x]] = x;
  }
  std::string res;
  for (int i = 0 ; i < L ; i++) {
    res += back_tr[best_pattern & 3];
    best_subsequence >>= 2;
  }
  std::reverse(res.begin(), res.end());
  return res;
}
```

C'est bien mieux, et surtout passe sans problème sur le serveur de test, bien plus rapide que la machine utilisée pour rédiger cette correction.

Nous encourageons les candidats à se familiariser avec les opérations bit-à-bit et avec leur syntaxe dans leur langage de programmation préféré.

3.2.3 Des solutions plus rapides, pour les curieux

Y a-t-il plus rapide ? Nous allons voir que oui. Notez que les solutions que nous présentons dans cette partie n'étaient absolument pas nécessaires pour obtenir le maximum de points à cet exercice.

Quelque part, la vitesse de la solution précédente nous laisse un sentiment d'inachevé. Nous avons gagné un facteur $L = 15$ en complexité théorique, et pourtant, nous sommes passés seulement de 250 ms à 84 ms, ce qui correspond environ à un facteur 3[4].

Une réponse partielle est que toutes les opérations de complexité théorique $O(1)$ (par exemple, l'accès à un élément dans une table de hachage) ne s'exécutent pas toutes à la même vitesse sur la machine.

Tout d'abord, il faut se demander quelles sont les opérations coûteuses dans notre programme des listings 4 page 7 et 5 page précédente. Si l'on n'a pas d'idées, il faut utiliser des outils comme `gprof` sur Linux et Shark sur Mac OS X. Ils nous indiquent que notre programme passe la grande majorité du temps à exécuter la ligne `++freqs[cur] ;`.

D'abord quelques mots sur le fonctionnement d'une table de hachage simple. Ce qu'il faut savoir est qu'accéder à un élément dans une table de hachage revient à accéder à une certaine case d'un tableau à un indice arbitraire calculé à partir de votre index. Notre solution passe donc son temps à accéder à des cases particulières d'un tableau avec la ligne `++freqs[cur] ;`.

En simplifiant un peu, chaque accès à la table de hachage prend 84 ms / 200000 = 42 µs. Sachant que notre microprocesseur fait 2 milliard de cycles par seconde (c'est le sens de 2 GHz !), chaque accès à la table de hachage occupe environ 840 cycles. C'est énorme sachant que chaque opération de base (par exemple, addition, décalage de bit, comparaison) prend 1 cycle.

L'explication est que le tableau sous-jacent à la table de hachage ne tient pas dans la mémoire cache du microprocesseur, et que chaque accès à la table de hachage requiert d'accéder à la mémoire centrale (RAM) de l'ordinateur. Chacun de ces accès est très coûteux, et le microprocesseur passe donc son temps à attendre la mémoire vive (soit en lecture soit en écriture). L'outil `valgrind` confirme nos dires :

```
$ valgrind --tool=cachegrind ./a.out  < test11.in
==17623== Cachegrind, a cache and branch-prediction profiler.
==17623== Copyright (C) 2002-2007, and GNU GPL'd, by Nicholas Nethercote et al.
==17623== Using LibVEX rev 1804, a library for dynamic binary translation.
==17623== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==17623== Using valgrind-3.3.0-Debian, a dynamic binary instrumentation framework.
==17623== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==17623== For more details, rerun with: -v
==17623==
ACACAGGTAGCAGGG
==17623==
==17623== I   refs:      96,535,045
==17623== I1  misses:         1,421
```

[4] Le temps pris par la lecture de l'entrée et ce qui n'est pas inclus dans la boucle principale est inférieur à 10 ms.

```
==17623== L2i misses:         1,405
==17623== I1  miss rate:       0.00%
==17623== L2i miss rate:       0.00%
==17623==
==17623== D   refs:      49,661,967  (32,019,854 rd   + 17,642,113 wr)
==17623== D1  misses:       920,347  (   850,890 rd   +     69,457 wr)
==17623== L2d misses:       388,159  (   318,860 rd   +     69,299 wr)
==17623== D1  miss rate:        1.8% (       2.6%     +        0.3%  )
==17623== L2d miss rate:        0.7% (       0.9%     +        0.3%  )
==17623==
==17623== L2 refs:          921,768  (   852,311 rd   +     69,457 wr)
==17623== L2 misses:        389,564  (   320,265 rd   +     69,299 wr)
==17623== L2 miss rate:         0.2% (       0.2%     +        0.3%  )
```

L'avant-dernière ligne montre le nombre de fois où les données ne sont pas présentes dans le cache[5]. On voit qu'il y a 320 265 défauts de cache en lecture (rd = read), et 69 299 en écriture (wr = write).

Une idée pour accélérer notre solution consiste donc à réduire le nombre de défauts de cache.

Une possibilité est d'utiliser une table de hachage plus rapide que celle fournie dans la bibliothèque standard du C++. Par exemple, utiliser la table de hachage `dense_hash_map` disponible sur `http://code.google.com/p/google-sparsehash/` nous a permis d'atteindre la vitesse suivante :

```
$ time ./a.out < test11.in
real 0m0.040s
user 0m0.032s
sys 0m0.004s
```

avec beaucoup moins de défauts de cache en lecture :

```
==20700== L2 misses:        240,134  (   171,050 rd   +    69,084 wr)
```

Une autre idée est beaucoup plus simple, et pourtant très efficace. Au lieu d'indexer une table associative par les nombres correspondant aux sous-chaînes de longueur L de la séquence d'ADN, ajoutons-les séquentiellement à un tableau. Ensuite, pour trouver l'entier le plus fréquent, trions le tableau et cherchons-le par un parcours linéaire du tableau trié. Avec cette idée et le programme dans le listing 6 page 12, l'on obtient les temps suivants :

```
$ time ./a.out < test11.in
real 0m0.026s
user 0m0.024s
sys 0m0.000s
```

Remarquez que c'est presque 4 fois plus rapide que la solution des listings 4 page 7 et 5 page 8 utilisant la table de hachage de la bibliothèque standard du C++, alors que la complexité de la nouvelle solution est en $O(N \log(N))$ contre $O(N)$. L'explication est simple avec `valgrind` :

```
$ valgrind --tool=cachegrind ./a.out  < test11.in
[...]
```

[5] *cache misses* en anglais

```
==20969==
ACACAGGTAGCAGGG
==20969==
==20969== I   refs:      39,882,697
==20969== I1  misses:         1,429
==20969== L2i misses:         1,385
==20969== I1  miss rate:       0.00%
==20969== L2i miss rate:       0.00%
==20969==
==20969== D   refs:      15,715,898  (11,967,531 rd   + 3,748,367 wr)
==20969== D1  misses:       139,672  (   120,124 rd   +    19,548 wr)
==20969== L2d misses:        23,561  (     4,440 rd   +    19,121 wr)
==20969== D1  miss rate:        0.8% (       1.0%     +       0.5%  )
==20969== L2d miss rate:        0.1% (       0.0%     +       0.5%  )
==20969==
==20969== L2 refs:          141,101  (   121,553 rd   +    19,548 wr)
==20969== L2 misses:         24,946  (     5,825 rd   +    19,121 wr)
==20969== L2 miss rate:         0.0% (       0.0%     +       0.5%  )
```

Cette solution a tout simplement réduit très fortement le nombre de défauts de cache, tant en lecture (5 825 contre 320 265 auparavant), qu'en écriture, qui sont les plus coûteux (19 548 contre 69 457 auparavant). La raison pour laquelle ces nombres baissent est que les motifs des accès à la mémoire lors d'un tri sont beaucoup plus prédictibles par le microprocesseur (qui précharge alors certaines zones de la mémoire vive dans sa mémoire cache) que des accès totalement aléatoires, comme c'était le cas de la solution avec une table de hachage.

Enfin, bien que la complexité théorique soit plus élevée, le nombre d'instructions executées par le microprocesseur (section `I refs`) a également été réduit de 95M à 40M.

Nous invitons ceux qui sont intéressés par toutes les problématiques relatives à la gestion de la mémoire à lire l'excellent article [1].

3.2.4 Les méthodes à retenir

Il faut également retenir de cet exercice qu'il faut toujours essayer d'utiliser toutes les hypothèses de l'énoncé, même cachées. Ici, c'est le fait que les chaînes n'utilisent que les quatre lettres A, T, G et C qui a principalement permis des solutions plus rapides que notre première solution.

Nous vous encourageons également à tester votre programme sur votre machine, en générant au besoin de gros tests. Enfin, pour les plus experts d'entre vous, sachez que la complexité théorique ne fait pas tout, et cache souvent des simplifications importantes ; par exemple, les motifs d'accès à la mémoire sont importants pour la vitesse d'exécution d'un programme.

3.3 Solutions proposées par les candidats

L'essentiel des candidats ont proposé une solution très similaire à celle qu'ils avaient proposé pour l'exercice 2 : souvent la même structure de données, consultée de manière assez naïve (en construisant une nouvelle sous-chaîne à chaque index). Seuls quelques-uns ont pensé à réutiliser la sous-chaîne commençant à l'index i pour construire celle commençant en $i + 1$.

Certains ont cependant trouvé l'astuce de coder le tout par des entiers et d'effectuer les décalages de bits. Ils ont alors eu tous les points. D'autres ont eu la bonne idée mais ne sont pas allés jusqu'au bout : probablement non familiers avec les représentations binaires, ils ont eu l'idée de coder les séquences par les entiers, mais en les calculant explicitement à l'aide d'une

Listing 6 : Deuxième solution pour l'exercice 3 en C++

```
std::string MostFrequentSubsequence(const std::string& adn_sequence, int L) {
  const int N = adn_sequence.size();
  std::vector<int> freqs;
  freqs.reserve(N);
  uint cur = 0;
  uint mask = (1<<(2*L)) - 1;

  [...] création de tr [...]

  for (int i = 0 ; i < N ; ++i) {
    cur <<= 2;
    cur |= tr[adn_sequence[i]];
    cur &= mask;
    if (i >= L - 1) {
      freqs.push_back(cur);
    }
  }

  std::sort(freqs.begin(), freqs.end());

  int max_freq = 0;
  uint best_pattern;
  int cur_freq = 1;
  for (int i = 1 ; i < freqs.size() ; i++) {
    if (freqs[i] == freqs[i-1]) {
      cur_freq++;
    } else {
      if (cur_freq > max_freq || cur_freq == max_freq
          && freqs[i-1] < best_pattern) {
        max_freq = cur_freq;
        best_pattern = freqs[i-1];
      }
      cur_freq = 1;
    }
  }

  // traduction de best_pattern en chaîne comme précédemment
}
```

fonction calculant les puissances de 4 ! Cela était inutilement lent (mais montrait qu'ils avaient quand même eu la bonne idée)…

Une solution partagée par plusieurs copies que nous n'avons pas expliquée ici est l'utilisation d'une trie (arbre où l'on descend d'un niveau à chaque fois que l'on lit un caractère). C'était une bonne idée, qui passait tous les tests sur le site d'entraînement, mais qui ne permet pas de s'affranchir du facteur $O(L)$.

On a également trouvé la solution contre-intuitive que nous venons d'expliquer consistant à trier le tableau ! Cependant, l'absence de tout commentaire dans ces copies suggère plutôt que les candidats en question n'avaient pas pris en compte les considérations exposées ci-dessus et n'avaient pas essayé la table de hachage…

4 Transformation

4.1 Énoncé

On vous donne en entrée deux séquences d'ADN de tailles n_1 et n_2. On souhaite calculer le nombre minimal de transformations pour passer de la première séquence à la deuxième. Les transformations possibles sont la modification, l'ajout ou la suppression d'un nucléotide. Les deux séquences sont très similaires ; on garantit donc que l'on puisse passer de l'une à l'autre en moins de 100 transformations. Ecrivez un programme qui renvoie ce nombre.

4.2 Solution

Commençons par résoudre le problème générique : une distance d'édition, aussi appelée distance de Levenshtein, classique, qui se résout par l'algorithme de Wagner-Fischer [2].

L'idée, comme souvent, est de formuler le problème de façon récursive : on va écrire une fonction $d(i, j)$ qui va renvoyer le nombre minimum de transformations pour rendre les sous chaînes $A[1..i]$ et $B[1..j]$ identiques. La solution est alors donnée par $d(n_1, n_2)$.

On a alors deux cas : soit $A[i] = B[j]$ et $d(i, j) = d(i-1, j-1)$, soit on a une lettre différente, et il faut alors choisir le moins coûteux entre :

- rendre égales $A[1..i-1]$ et $B[1..j]$ et supprimer la lettre $A[i]$: cela a un coût de $d(i-1, j)+1$;
- rendre égales $A[1..i]$ et $B[1..j-1]$ et insérer la lettre $B[j]$: cela a un coût de $d(i, j-1)+1$;
- rendre égales $A[1..i-1]$ et $B[1..j-1]$ et changer $A[i]$ en $B[j]$: cela a un coût de $d(i-1, j-1) + 1$.

On peut donc calculer d récursivement, et comme d'habitude, pour ne pas refaire les mêmes calculs plusieurs fois, stocker les résultats dans un tableau.

Pour utiliser moins de mémoire et satisfaire aux contraintes de mémoire du site d'entraînement, il est possible de ne conserver qu'une ligne de la matrice en cours de calcul.

Cependant, tout cela était encore trop lent. Sur le site d'entraînement, on indiquait $n_1, n_2 \leq 10^5$, soit un ordre de grandeur de 10^{10} opérations (la complexité étant $O(n_1 n_2)$).

Il faut donc utiliser l'hypothèse donnée par l'énoncé : *on garantit donc que l'on puisse passer de l'une à l'autre en moins de 100 transformations*. On voit alors qu'il est inutile de calculer $d(i, j)$ si $|i-j| > 100$. C'est à dire qu'on ne va calculer les valeurs qu' « autour de la diagonale » .

Avec cette hypothèse capitale, on obtient le code du listing 7 page suivante, qui est celui que nous attendions.

Il est à noter que l'on peut améliorer l'occupation mémoire de cet algorithme en calculant les valeurs du tableau de manière itérative ligne par ligne, comme nous l'avons mentionné.

Listing 7 : Solution de l'exercice 4 en C++

```
const int maxN = 10000;
const int maxD = 100;
const int inf = 1000000000;

int dyna[maxN][maxD*2+10];
// initialisé à -1 dans la fonction principale

int editDistance(int i, int j, const string& u, const string& v) {
  if (i == u.size())
    return v.size() - j;
  if (j == v.size())
    return u.size() - i;
  if (abs(i-j) > maxD) return inf;
  int &res = dyna[i][i-j+maxD];
  // +maxD permet de translater [-maxD,maxD] en [0, 2 * maxD]
  if (res >= 0) return res;
  res = inf;
  res = min(res, editDistance(i+1, j, u, v)+1);
  res = min(res, editDistance(i, j+1, u, v)+1);
  res = min(res, editDistance(i+1, j+1, u, v) + !(u[i] == v[j]));
  return res;
}
```

4.3 Solutions proposées par les candidats

Lorsque cet exercice a été abordé, il a été fait par la majorité des candidats selon une méthode ne prenant pas en compte l'hypothèse $|i - j| \leq 100$. Seuls les meilleurs ont pensé à l'utiliser, et c'était nécessaire pour avoir tous les points. Ce n'était pourtant pas très compliqué lorsqu'on avait bien compris le principe de la solution de base ; le code correspondant n'en différait d'ailleurs que par quelques lignes.

Ne pas prendre en compte cette hypothèse ne faisait en outre pas passer tous les tests sur le serveur d'entraînement, sauf dans le cas de solutions particulièrement bien écrites. Le nom « distance de Levenshtein » ayant circulé sur le forum, l'on peut donc en déduire qu'un grand nombre de candidats, en cet âge de l'Internet, ont trouvé l'algorithme sur Wikipédia ou un site Web quelconque et l'ont recopié ou traduit dans leur langage. Ce n'était pas le but du concours, mais nous avons quand même accordé à ceux-là le bénéfice du doute en leur donnant la moitié des points.

Merci à tous d'avoir participé et bonnes révisions à tous les sélectionnés pour les épreuves régionales ! Et bon courage aux recalés, vous pourrez retenter votre chance en 2011 après avoir travaillé un peu…

Références

[1] Ulrich Drepper : What every programmer should know about memory, 2007. `http://people.redhat.com/drepper/cpumemory.pdf`.

[2] Robert A. Wagner et Michael J. Fischer : The string-to-string correction problem. *Journal of the ACM*, 21(1) :168–173, 1974.