Cherchez et vous trouverez,
. . . car qui cherche trouve.
Matthieu **7** 7-8 et Luc **11** 9-10

CHAPITRE V

Recherche et tri

Objectif. Comprendre les techniques de base pour organiser des données ordonnées.

Ce chapitre discute quelques méthodes de recherche et de tri. Les applications sont abondantes ; imaginez par exemple à quel point un dictionnaire serait difficile à utiliser si les mots clés n'étaient pas ordonnés ! Ils le sont, heureusement, car l'éditeur a pris le soin de les *trier*. Vous en profitez en appliquant une méthode efficace de *recherche*. Le projet à la fin de ce chapitre illustre une application moins évidente : la recherche des solutions d'une équation diophantienne (par exemple $x^4+y^4+z^4=w^4$ avec $x,y,z,w\in\mathbb{Z}_+$).

Sommaire

Exemple 0.1. Les deux problèmes fondamentaux, tri et recherche, peuvent se formuler ainsi :

(1) Trier une liste donnée afin d'établir l'ordre $a_1 \le a_2 \le \cdots \le a_n$.

(2) Chercher un élément b dans une liste ordonnée $(a_1 \le a_2 \le \cdots \le a_n)$.

Ajoutons que le tri résout bien d'autres problèmes concernant les listes, a priori non triées :

(3) Trouver les éléments doubles dans une liste (problème de multiplicité).

(4) Vérifier si deux listes sont les mêmes à permutation près (problème d'anagramme).

(5) Trouver la médiane d'une longue liste de valeurs réelles (problème de rang).

Remarque 0.2. D.E. Knuth dans *The Art of Computer Programming* discute diverses méthodes de tri dans le tome 3, intitulé *Sorting and Searching*. Tout au début il fait le constat suivant, toujours valable de nos jours :

> Computer manufacturers estimate that over 25% of the running time on their computers is currently being spent on sorting (. . .) We may conclude that (i) there are many important applications of sorting, or (ii) many people sort when they shouldn't, or (iii) inefficient sorting algorithms are in common use. The real truth probably involves some of all three alternatives. In any event we can see that sorting is worthy of serious study, as a practical matter.

Ce chapitre tente d'expliquer puis de comparer différentes méthodes de recherche (§1) et de tri (§2–§3). Les exercices mathématiques permettront d'établir la correction des méthodes proposées et de comparer leur performance. Cette partie peut sembler un peu théorique mais elle est très bénéfique pour comprendre ce qu'est l'algorithmique. La morale à retenir : préférer les bons algorithmes aux solutions naïves !

☞ Si vous êtes impatient ou insouciant, vous pouvez lire la recherche dichotomique (algorithme V.2) puis le tri fusion (algorithme V.6) et le tri rapide (algorithme V.7), afin de passer directement à l'implémentation (§4). En §5 nous regardons brièvement quelles solutions offre la STL.

1. Recherche linéaire vs recherche dichotomique

Dans la pratique les données à chercher ou à trier peuvent être des nombres ou des chaînes de caractères ou bien d'autres objets encore. Les méthodes présentées ici s'étendent sans aucune modification aux éléments d'un ensemble ordonné E quelconque. Ceci veut dire que E est muni d'une relation d'ordre $<$ de sorte que pour tous $a, b \in E$ on ait ou $a < b$ ou $a = b$ ou $b < a$ (trichotomie), et que $a < b$ et $b < c$ implique $a < c$ (transitivité). Étant donné un tel ordre $<$ on définit les relations $\leq$, $>$, $\geq$ comme d'habitude, afin d'avoir une écriture plus commode.

1.1. Recherche linéaire. Pour commencer nous allons considérer le problème de la recherche d'un élément dans une liste $A = (a_1, a_2, \ldots, a_n)$. Étant donné un élément b, on souhaite déterminer s'il appartient à la liste A, et si oui, trouver le premier indice k tel que $a_k = b$. La méthode évidente consiste à parcourir toute la liste jusqu'à atteindre l'élément cherché :

Algorithme V.1 Recherche linéaire

Entrée: Un élément b et une liste $A = (a_1, a_2, \ldots, a_n)$.

Sortie: Le premier indice k tel que $a_k = b$, ou bien *non trouvé* si b n'appartient pas à A.

```
pour k de 1 à n faire
  si a_k = b alors retourner k
fin pour
retourner non trouvé
```

Exercice/M 1.1. Expliquer pourquoi la recherche linéaire nécessite n itérations dans le pire des cas, une itération seulement dans le meilleur des cas, et $\frac{n+1}{2}$ itérations en moyenne (en supposant que l'on choisit b dans A au hasard). Pourquoi cette recherche est-elle appelée *linéaire* ?

1.2. Recherche dichotomique. La recherche linéaire se révèle assez inefficace si le nombre d'éléments dans la liste est élevé. La recherche peut être considérablement accélérée si la liste est ordonnée dans le sens que $a_1 \leq a_2 \leq \cdots \leq a_n$. La méthode suivante est un exemple classique du paradigme *diviser pour régner* : on compare d'abord l'élément cherché b avec a_m au milieu de la liste, puis on continue la recherche sur la moitié adéquate de la liste.

Algorithme V.2 Recherche dichotomique

Entrée: Un élément b et une liste ordonnée $A = (a_1, a_2, \ldots, a_n)$.

Sortie: Le premier indice k tel que $a_k = b$, ou bien *non trouvé* si b n'appartient pas à A.

```
i ← 1,  j ← n                          // Si A contient b, alors le premier indice est entre i et j.
tant que i < j faire
  m ← ⌊(i+j)/2⌋                        // calculer l'indice au milieu, arrondi arbitrairement
  si b ≤ a_m alors j ← m sinon i ← m+1  // continuer avec la moitié gauche ou droite
fin tant que
si a_i = b alors retourner i sinon retourner non trouvé
```

Exemple 1.2. Pour voir le fonctionnement de l'algorithme V.2 sur un exemple concret, faites le tourner « à la main » sur la liste ordonnée suivante :

$$(1) \qquad (12, 17, 20, 34,\ 37, 37, 36, 42,\ \ 48, 57, 61, 64,\ 67, 70, 77, 94).$$

Chercher ainsi la première occurrence de 61, puis 37 (qui y figure deux fois), finalement 50 (qui n'y est pas). Pour un exemple plus réaliste vous pouvez ensuite regarder une liste dont la taille n'est pas une puissance de 2, et vérifier que le principe s'applique pareil.

Exercice/M 1.3. Prouver que l'algorithme V.2 est correct. Pour ce faire montrer d'abord qu'il se termine : la différence $j - i$ est un entier positif ou nul qui diminue à chaque itération jusqu'à ce que $i = j$. Vérifier ensuite que chaque itération préserve la propriété suivante : si A contient b, alors le premier indice k tel que $a_k = b$ se trouve dans l'intervalle $[\![i, j]\!]$. Conclure.

Exemple 1.4. Comme dans l'exemple précédent, regardons une liste de longueur 16 mais contenant cette fois-ci les nombres binaires $\texttt{0000}_{\text{bin}},\dots,\texttt{1111}_{\text{bin}}$. Vérifiez que la première itération de la recherche dichotomique divise cette liste en deux partie, à savoir $(\texttt{0000}_{\text{bin}},\dots,\texttt{0111}_{\text{bin}})$ et $(\texttt{1000}_{\text{bin}},\dots,\texttt{1111}_{\text{bin}})$. On détermine ainsi le premier bit. La deuxième itération détermine le deuxième bit, et ainsi de suite. Pour cette raison la recherche dichotomique est aussi appelée *recherche binaire*, ou *binary search* en anglais.

Exercice/M 1.5. Pour $n = 2^k$ vérifier que l'algorithme V.2 nécessite exactement k itérations : initialement l'intervalle $[\![i, j]\!]$ contient 2^k éléments, et chaque itération divise la longueur par deux. Plus généralement, montrer le théorème suivant. Justifier ainsi pourquoi la recherche dichotomique est préférable à la recherche linéaire, voire indispensable si n est grand.

Théorème 1.6. *Pour trouver un élément dans une liste ordonnée de longueur n, la recherche dichotomique définie par l'algorithme V.2 nécessite* $\lceil \log_2 n \rceil$ *ou* $\lfloor \log_2 n \rfloor$ *itérations seulement.*

Remarque 1.7. Comme une variante de la recherche dichotomique vous pouvez reprendre le jeu « trop petit, trop grand » esquissé en chapitre I, exercices 8.6 et 8.7. Pour trouver un nombre dans $[\![1, 127]\!]$ explicitez une stratégie qui ne nécessite que 6 questions. Explicitez ensuite une stratégie dans le cas général. Expliquez pourquoi il s'agit ici d'une recherche « trichotomique » plutôt que « dichotomique ».

1.3. Attention aux détails ! Rappelons que le but d'un algorithme est de préciser sans aucune ambiguïté le procédé à exécuter. L'avantage d'une formulation précise est, bien évidemment, que l'on puisse établir sa correction une fois pour toutes, pour ensuite l'utiliser la conscience tranquille. Pour cette raison il faut faire attention au moindre détail ; toute erreur, même minuscule, peut s'avérer catastrophique dans des futures implémentations :

Exercice/M 1.8. Dans l'algorithme V.2, si l'on remplaçait la condition $i < j$ par la condition $i \leq j$, l'algorithme modifié serait-il correct ? Donner une preuve ou un contre-exemple.

Exercice/M 1.9. Dans l'algorithme V.2, si l'on remplaçait la comparaison $b \leq a_m$ par $b < a_m$, l'algorithme modifié serait-il correct ? Donner une preuve ou un contre-exemple.

Exercice/M 1.10. Dans l'algorithme V.2, si l'on remplaçait $m \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$ par $m \leftarrow \left\lceil \frac{i+j}{2} \right\rceil$, l'algorithme modifié serait-il correct ? Donner une preuve ou un contre-exemple.

2. Trois méthodes de tri élémentaires

Étant donnée une famille $A = (a_1, a_2, \dots, a_n)$ d'éléments dans un ensemble ordonné E, le but d'un tri est de déterminer une permutation $\sigma\colon \{1,2,\dots,n\} \to \{1,2,\dots,n\}$ qui mette les éléments en ordre croissant, c'est-à-dire $a_{\sigma(1)} \leq a_{\sigma(2)} \leq \cdots \leq a_{\sigma(n)}$.

Dans ce qui suit, nous regarderons la variante suivante : on commence par une famille A stockée sous forme d'un tableau, et on souhaite le permuter de sorte que le tableau A' qui en résulte soit ordonné. Afin d'économiser la mémoire et le temps de copie, c'est un seul tableau qui est utilisé, qui au début représente A et qui finit par représenter A'. Ainsi on ne construira pas explicitement la permutation σ, mais directement le résultat de son action sur le tableau A.

2.1. Les plus petits exemples. Avant de discuter des méthodes plus générales, il semble utile de regarder le tri de deux, trois, ou quatre éléments.

Exercice 2.1. Commençons par le plus petit exemple, le tri de deux éléments, de type `int` disons :

```
void trier( int& a, int& b ) { if ( a > b ) { int c=a; a=b; b=c; }; };
```

Expliquer le fonctionnement de cette fonction en distinguant les deux configurations initiales possibles, à savoir $a \leq b$ et $a > b$. La fonction est-elle correcte ? Expliquer l'importance du symbole ' `&` ' dans la liste des paramètres.

Remarque 2.2. Comme l'opération d'échanger deux éléments sera omniprésente dans la suite, il est plus commode de définir une fonction qui effectue ce travail répétitif. On pourrait donc écrire :

```
void echanger( int& a, int& b ) { int c=a; a=b; b=c; };
void trier( int& a, int& b ) { if ( a > b ) echanger(a,b); };
```

Dans cet exemple l'écriture est plus longue, mais elle devient économique à partir de trois éléments.

Exercice 2.3. Après deux, regardons le tri de trois éléments. En voici un candidat :

```
void trier( int& a, int& b, int& c )
{ if ( a > b ) echanger(a,b);
  if ( b > c ) echanger(b,c);
  if ( a > b ) echanger(a,b); };
```

Expliquer le fonctionnement de cette fonction en distinguant toutes les six configurations initiales possibles, allant de $a \leq b \leq c$ jusqu'à $c < b < a$. La fonction est-elle correcte ?

Exercice 2.4. Remarquons que la fonction précédente nécessite toujours 3 comparaisons et effectue entre 0 et 3 échanges. Peut-on trouver une fonction plus efficace ? Voici un candidat :

```
void trier( int& a, int& b, int& c )
{ if ( a > b ) echanger(a,b);
  if ( b > c ) { echanger(b,c); if ( a > b ) echanger(a,b); }; };
```

Expliquer le fonctionnement de cette fonction puis montrer sa correction. Vérifier que cette fonction nécessite le même nombre d'échanges, mais seulement entre 2 et 3 comparaisons. Quel est le nombre moyen de comparaisons si toutes les six configurations initiales sont équiprobables ?

Remarque 2.5. Pour $n \geq 4$ éléments on souhaiterait sans doute une méthode générale de tri, ce que nous discuterons dans les paragraphes suivants. Néanmoins on peut s'intéresser aux méthodes *optimales* de tri, pour une taille n donnée. Une autre façon de ce voir est de regarder un tournoi de n joueurs, que l'on veut classer dans l'ordre de performance (la liste triée) avec un nombre minimum de matchs (comparaisons). Si cette question vous intéresse, consultez Knuth [**8**], §5.3.1.

Exercice 2.6. Écrire une fonction optimale pour trier quatre éléments. Est-ce possible en effectuant au plus 4 comparaisons ? au plus 5 comparaisons ? (Contempler l'inégalité $2^4 < 4! < 2^5$.)

Exercice 2.7. Essayez d'écrire une fonction optimale pour trier cinq éléments. Est-ce possible en effectuant au plus 6 comparaisons ? au plus 7 comparaisons ? (Contempler $2^6 < 5! < 2^7$.)

Exercice 2.8. Avant de continuer, expliquez par quelle méthode vous triez dans la vie quotidienne. Par exemple, comment triez-vous une main de cartes ? Appliquez votre méthode à la famille

$$(7,6,1,3,1,7,2,4) \tag{2}$$

Il y a des fortes chances que votre méthode soit une des trois suivantes :

2.2. Tri par sélection. C'est sans doute la méthode de tri la plus élémentaire. On cherche le plus petit élément a_m parmi $a_1, a_2, \ldots, a_n$, puis on le place au début de la liste. Cette opération est répétée pour la liste restante, jusqu'à ce que tous les éléments soient dans l'ordre.

Exercice 2.9. L'algorithme V.3 réalise un tri par sélection. Montrer sa correction, c'est-à-dire montrer qu'il produit bien une suite ordonnée comme énoncé.

Algorithme V.3 Tri par sélection (*selection sort* en anglais)

Entrée: Une famille $(a_1, a_2, \ldots, a_n)$ d'éléments d'un ensemble ordonné.

Sortie: La même famille, permutée de sorte que $a_1 \leq a_2 \leq \cdots \leq a_n$.

pour i **de** 1 **à** $n-1$ **faire**
  $m \leftarrow i$
  pour j **de** $i+1$ **à** n **faire**
    si $a_j < a_m$ **alors** $m \leftarrow j$
  fin pour
  échanger $a_i \leftrightarrow a_m$
fin pour

Exercice/M 2.10. Essayons d'estimer plus précisément le coût de cet algorithme. Supposons que chaque itération de la boucle intérieure nécessite un temps α_1, et que chaque itération de la boucle extérieure ajoute un coût β_1. Vérifier que le coût total de l'algorithme V.3 est alors $c_1(n) = \frac{1}{2}\alpha_1 n(n-1) + \beta_1(n-1)$. Pour n grand, expliquer pourquoi c'est le terme dominant $\frac{1}{2}\alpha_1 n^2$ qui l'emporte.

2.3. Tri par transposition. Une variante du tri par sélection est le tri par transposition, aussi appelé le *tri bulle*. On regarde si deux éléments voisins sont dans le mauvais ordre, si oui on les échange. Cette opération est itérée jusqu'à ce que tous les éléments soient dans l'ordre.

Exercice 2.11. L'algorithme V.4 réalise un tri par transposition. Montrer sa correction. Combien d'itérations faut-il ? Vérifier que son coût est $c_2(n) = \frac{1}{2}\alpha_2 n(n-1) + \beta_2(n-1)$. Pouvez-vous trouver des améliorations ?

Algorithme V.4 Tri par transposition (tri bulle ou *bubble sort* en anglais)

Entrée: Une famille $(a_1, a_2, \ldots, a_n)$ d'éléments d'un ensemble ordonné.

Sortie: La même famille, permutée de sorte que $a_1 \leq a_2 \leq \cdots \leq a_n$.

pour i **de** 1 **à** $n-1$ **faire**
  pour j **de** 1 **à** $n-i$ **faire**
    si $a_j > a_{j+1}$ **alors** échanger $a_j \leftrightarrow a_{j+1}$
  fin pour
fin pour

2.4. Tri par insertion. C'est le tri du joueur de cartes. On considère que les éléments de la liste à trier sont donnés un par un. Avant de recevoir l'élément a_i on a déjà trié les éléments $a_1, \ldots, a_{i-1}$. On insère alors l'élément a_i à la position appropriée. Après cette opération la liste $\overline{a_1}, \ldots, \overline{a_i}$ est ordonnée, et on itère jusqu'à ce que tous les éléments soient dans l'ordre.

Exercice 2.12. L'algorithme V.5 réalise un tri par insertion. Montrer sa correction. Combien d'itérations faut-il en moyenne ? Vérifier que son coût moyen est $c_3(n) = \frac{1}{4}\alpha_3 n(n-1) + \beta_3(n-1)$. Peut-on déjà conclure que cet algorithme est supérieur aux précédents ? Quel est l'intérêt des comparaisons empiriques ?

Algorithme V.5 Tri par insertion (*straight insertion sort* en anglais)

Entrée: Une famille $(a_1, a_2, \ldots, a_n)$ d'éléments d'un ensemble ordonné.

Sortie: La même famille, permutée de sorte que $a_1 \leq a_2 \leq \cdots \leq a_n$.

pour i **de** 2 **à** n **faire**
  $a \leftarrow a_i$, $j \leftarrow i-1$
  tant que $j > 0$ et $a_j > a$ **faire** $a_{j+1} \leftarrow a_j$, $j \leftarrow j-1$
  $a_{j+1} \leftarrow a$
fin pour

2.5. En sommes-nous contents ? Les méthodes de tri discutées dans ce paragraphe sont assez simples et elles suffisent pour trier les petites listes. Pourtant, dans la pratique il faut souvent trier des listes assez grandes, disons de 10^6 éléments, voire beaucoup plus. (On rencontrera de tels exemples dans le projet V.)

Pour tester dans quelles limites nos méthodes sont praticables, vous pouvez les tester sur des données réelles, par exemple sur des listes « aléatoires ». Pour ceci vous pouvez utiliser la fonction suivante :

```
#include <cstdlib>
void aleatoire( vector<int>& vec  )
{ for ( int i=0; i<vec.size(); ++i ) vec[i]= random()%vec.size(); };
```

Exercice/P 2.13. Implémenter une ou plusieurs des méthodes de tri présentées ci-dessus. Laquelle des méthodes vous semble la plus simple à implémenter et/ou la plus efficace ? Les tester sur des listes aléatoires. Arrive-t-on ainsi à trier une liste de longueur 10^2 ? 10^3 ? 10^4 ? 10^5 ? 10^6 ?

Exemple 2.14. Regardons un annuaire comme les pages blanches (`www.pages-blanches.fr`) qui stocke quelques millions d'entrées, disons 10^8 éléments. Une méthode de tri de complexité quadratique y mettra environ 10^{16} itérations. Même à une vitesse foudroyante de 10^8 itérations par seconde, l'annuaire ne sera prêt que dans trois ans – et déjà dépassé.

Exercice 2.15. Pour un exemple encore plus impressionnant pensez à Google (`www.google.fr`) qui se dit de stocker quelques milliards d'entrées. Estimer la durée d'un tri avec une des méthodes ci-dessus.

3. Diviser pour trier

Les exemples précédents nous mène à la conclusion qu'une méthode de complexité quadratique n'est pas acceptable pour trier de grands fichiers. Les méthodes présentées ci-dessus sont donc inappropriées. Existe-t-il des méthodes de tri qui soient plus efficaces ? Heureusement que oui ! Ceci fait l'objet de ce paragraphe.

L'idée géniale de la recherche dichotomique est de diviser le problème en deux sous-problèmes de taille moitié. C'est le paradigme de « diviser pour régner ». Vu la réussite de cette approche on essaiera de faire pareil pour le tri. Cette idée a donné lieu à un bon nombre d'algorithmes de tri, dont on ne présentera que deux : le tri fusion et le tri rapide.

3.1. Le tri fusion. L'idée du tri fusion est simple : on divise en deux moitiés la liste à trier, on trie chacune d'elles, puis on fusionne les deux moitiés pour reconstituer la liste complète triée.

Algorithme V.6 Tri fusion (*merge sort* en anglais)

Entrée: Une famille $A = (a_1, a_2, \ldots, a_n)$ d'éléments d'un ensemble ordonné.

Sortie: La même famille, permutée de sorte que $a_1 \leq a_2 \leq \cdots \leq a_n$.

si $n \leq 1$ **alors retourner** A	// Si $n = 0$ ou $n = 1$ il n'y a rien à trier.
$p \leftarrow \lfloor n/2 \rfloor$, $q \leftarrow \lceil n/2 \rceil$	// décomposer n en deux moitiés p et q
$B = (b_1, \ldots, b_p) \leftarrow (a_1, \ldots, a_p)$	// produire une copie de la moitié gauche
$C = (c_1, \ldots, c_q) \leftarrow (a_{p+1}, \ldots, a_n)$	// produire une copie de la moitié droite
trier la liste B et la liste C	// on sait déjà trier des listes de taille p et q
fusionner B et C dans une liste triée A	// voir exercice 3.1 ci-dessous
retourner A	

Exercice 3.1. Compléter l'algorithme en explicitant comment fusionner les listes B et C dans une seule liste triée A avec n itérations seulement. On pourra traiter B et C comme deux « files d'attente ». Votre algorithme ainsi complété est-il correct ? Comment assure-t-il la terminaison ?

Exemple 3.2. Pour voir comment fonctionne l'algorithme, appliquez-le au tri de la liste (2).

Exercice 3.3. En appliquant le tri fusion à quatre équipes, montrer qu'il existe un tournoi qui ne nécessite que cinq matchs pour établir le classement. Expliquer pourquoi on ne peut pas faire mieux.

3.2. Analyse de complexité. En quoi l'algorithme V.6 est-il intéressant ? Pour y répondre il faut analyser son coût, c'est-à-dire le temps nécessaire pour son exécution. Soit $c(n)$ le coût de l'algorithme V.6 quand on l'applique à une liste de longueur n. Ceci comprend : (1) le coût pour diviser la liste en deux moitiés, (2) le coût pour trier les deux parties, et (3) le coût pour fusionner les deux listes en une seule.

Les étapes (1) et (3) nécessitent chacune n itérations ; leur coût est linéaire en n, disons αn avec une certaine constante $\alpha \geq 0$. L'étape (2) nécessite le tri de la moitié gauche B de taille $p = \lfloor \frac{n}{2} \rfloor$ et de la moitié droite C de taille $q = \lceil \frac{n}{2} \rceil$. S'ajoute un coût constant $\beta \geq 0$ pour le calcul de p et de q etc. Au total, pour $n \geq 2$, on arrive à un coût

$$c(n) = c(\left\lfloor \frac{n}{2} \right\rfloor) + c(\left\lceil \frac{n}{2} \right\rceil) + \alpha n + \beta \tag{3}$$

Exercice/M 3.4. Pour les puissances $n = 2^k$ vérifier les premières valeurs de $c(n)$:

n	1	2	4	8	16	32
$c(n)$	0	$2\alpha + \beta$	$8\alpha + 3\beta$	$24\alpha + 7\beta$	$64\alpha + 15\beta$	$160\alpha + 31\beta$

On devine déjà que la croissance est moins que quadratique. Plus précisément, pour $n = 2^k$ vous pouvez montrer par récurrence que $c(n) = \alpha n \log_2 n + \beta(n-1)$. Par une variante de cette récurrence, essayez de prouver le théorème suivant :

Théorème 3.5. *Soient $\alpha, \beta \geq 0$. Avec la valeur initiale $c(1) = 0$ la relation de récurrence* (3) *définit une fonction $c\colon \mathbb{Z}_+ \to \mathbb{R}$. Pour tout $n \in \mathbb{Z}_+$ cette fonction vérifie l'inégalité*

$$\alpha n \lfloor \log_2 n \rfloor + \beta(n-1) \quad \leq \quad c(n) \quad \leq \quad \alpha n \lceil \log_2 n \rceil + \beta(n-1).$$

Remarque 3.6. Si $\alpha, \beta > 0$, alors le terme dominant dans cette majoration est $\alpha n \lceil \log_2 n \rceil$, qui croît un peu plus rapidement que le terme linéaire $\beta(n-1)$. On conclut que pour n grand le tri fusion a un coût d'ordre $n \log_2 n$. Ceci est une amélioration énorme vis-à-vis un coût quadratique n^2. Le justifier, par exemple en traçant les deux fonctions ; les évaluer pour $n = 10^2, 10^3, 10^4, 10^5, 10^6$.

Remarque 3.7. Le tri fusion est fait sur mésure pour les listes. Dans ce cas la repartition en deux sous-listes et la fusion en une liste finale ne sont que des manipulations très efficaces de pointeurs et ne nécessitent pas de copier les objets eux-mêmes. En particulier, le tri fusion appliqué sur les listes ne nécessite pas de mémoire temporaire auxiliaire. (Essayez de le détailler si vous connaissez les listes.)

Quand on l'applique sur des vecteurs, par contre, le tri fusion nécessite beaucoup de copie inutiles, ce qui est coûteux en temps et en mémoire. L'algorithme est toujours correct, mais les vecteurs ne sont visiblement pas la structure des données optimale pour cette méthode. Dans une application réaliste il faut parfois tenir compte de cette restriction et du couplage étroit entre algorithme et structure de données.

3.3. Le tri rapide. Le tri fusion discutée ci-dessus est une méthode lucide et éprouvée. Son seul inconvenient est la copie des deux moitiés dans des listes auxiliaires. Le tri rapide ci-dessous évite de telles copies, il est plus rapide *en moyenne*, mais malheureusement sa performance est moins prévisible.

À nouveau l'idée de cet algorithme consiste à séparer en deux la liste initiale. Cette fois-ci on réorganise la liste afin d'obtenir deux parties $B = (a_1, \dots, a_q)$ et $C = (a_p, \dots, a_n)$ de sorte que $a_i \leq a_j$ pour tout $i \in [\![1, q]\!]$ et $j \in [\![p, n]\!]$. La séparation en deux listes se fait à l'aide d'un *pivot* a_m, choisi arbitrairement : on place dans la première partie de la liste les éléments plus petits que le pivot et dans la seconde les éléments plus grands que le pivot. Il suffit ensuite de trier les parties B et C séparément pour arriver à une liste complètement triée.

Algorithme V.7 Tri rapide (*quicksort* en anglais)

Entrée: Une famille $(a_1, a_2, \dots, a_n)$ d'éléments d'un ensemble ordonné.

Sortie: La même famille, permutée de sorte que $a_1 \leq a_2 \leq \cdots \leq a_n$.

```
si n > 1 alors
   m ← ⌊(n+1)/2⌋,  pivot ← a_m          // choisir un pivot, ici on prend l'élément au milieu
   p ← 1,  q ← n                        // p parcourt de gauche à droite, q de droite à gauche
   répéter
      tant que a_p < pivot faire p ← p+1     // trouver le premier élément a_p trop grand
      tant que a_q > pivot faire q ← q−1     // trouver le dernier élément a_q trop petit
      si p > q alors terminer la boucle      // condition d'arrêt anticipé
      si p < q alors échanger a_p ↔ a_q      // échanger a_p et a_q pour qu'ils soient dans l'ordre
      p ← p+1,  q ← q−1                      // faire avancer les indices
   jusqu'à p > q
   trier (a_1, ..., a_q) puis (a_p, ..., a_n)   // appliquer le tri rapide à chacune des deux parties
fin si
```

Exemple 3.8. Pour avoir une idée de son fonctionnement, faites tourner le tri rapide sur les listes de longueur 2 (deux configurations initiales) puis sur les listes de longueur 3 (six configurations initiales). Finalement, pour un exemple plus réaliste, appliquet-le à la liste (2).

Exercice/M 3.9. Montrer que l'algorithme V.7 est correct, c'est-à-dire expliquer pourquoi il produit bien une liste ordonnée comme il le prétend. *Indication.* — Évidemment la partie délicate est la boucle qui réorganise la liste en séparant la partie basse et la partie haute.

(1) Montrer d'abord que chaque itération de la boucle incrémente p et/ou décrémente q. Par conséquent la boucle se termine après un nombre fini d'itérations en assurant $p > q$.

(2) Reste à comprendre les opérations effectuées sur la liste : montrer par récurrence que la boucle garantie toujours l'inégalité $a_i \leq a_j$ pour (i, j) vérifiant $1 \leq i < p$ et $q < j \leq n$.

Comme on finit par $p > q$, on obtient alors deux sous-listes $(a_1, \dots, a_q)$ et $(a_p, \dots, a_n)$ de sorte que $a_i \leq a_j$ pour tout $i \in [\![1, q]\!]$ et $j \in [\![p, n]\!]$, comme souhaité. Conclure.

3.4. Analyse de complexité. L'approche « diviser pour régner » n'est efficace que si les deux sous-problèmes sont de tailles à peu près égales. Pour le tri rapide il est donc souhaitable que le pivot soit la *médiane*. Malheureusement nous ne disposons pas de méthode rapide pour trouver la médiane ; on choisit donc le pivot au milieu, en espérant le mieux. (On pourrait aussi bien prendre a_1 ou a_n ou a_m avec m un indice aléatoire entre 1 et n.) Il se peut, bien sûr, que le pivot soit mal choisi ; dans le pire des cas une des deux sous-listes ne contient qu'un seul élément et l'autre en contient $n-1$. Heureusement, ce cas arrive assez rarement. (Dans un certain sens le tri rapide préfère les données aléatoires, mais il peut bloquer sur des listes avec un méchant ordre initial.) Plus précisément on peut montrer :

Théorème 3.10. *Le tri rapide donné par l'algorithme V.7 est correct dans le sens qu'il trie toute liste donnée d'une longueur n quelconque. Dans le pire des cas cet algorithme nécessite* $\frac{1}{2}n^2$ *itérations environ, mais en moyenne il n'en nécessite que* $2n\ln n$. □

Malgré ses inconvénients, beaucoup de programmeurs préfèrent le tri rapide, car il est facile à implémenter et évite toute copie dans des listes auxiliaires. Avec une implémentation optimisée, il est (en moyenne !) entre 30% et 50% plus rapide que d'autres méthodes de tri. Pour une analyse détaillée et des améliorations possibles voir Knuth [**8**], §5.2.2, ou Sedgewick [**7**], chapitre 9.

4. Fonctions génériques

4.1. Les calamités de la réécriture inutile. Dans les exemples du paragraphe 2 nous avons traité le tri d'entiers. Dans un autre contexte vous voulez peut-être trier des chaînes de caractères, puis des nombres de type `double`, etc... Après réflexion, le problème de tri est toujours le même ! Cependant, pour notre implémentation en C++, nous devons spécifier le type. Que faire ?

Une façon répandue de réaliser ces différentes fonctions est, hélas, l'usage de « copier-coller » puis le remplacement « à la main » de `int` par `string`, et la semaine prochaine de `string` par `double` etc. Supposons que la semaine d'après vous découvrez une erreur cachée dans votre méthode de tri. Vous corrigez alors trois variantes : la variante `int`, la variante `string` puis la variante `double`. Après réflexion vous voulez améliorer votre méthode de tri, donc vous changez à nouveau trois variantes... Il va sans dire que cette approche se révèlera infernale !

4.2. L'usage des fonctions génériques. Heureusement le C++ prévoit un concept qui permet de mieux organiser la programmation, économiser du temps, et éviter les fautes de frappes d'un recopiage erroné. Dans notre exemple, on veut éviter d'écrire plusieurs fonctions quasi identiques :

```
void echanger(    int& a,    int& b ) {    int c=a; a=b; b=c; }
void echanger( string& a, string& b ) { string c=a; a=b; b=c; }
void echanger( double& a, double& b ) { double c=a; a=b; b=c; }
```

Il sera beaucoup plus économique d'écrire une seule fonction générique :

```
template <typename T>
void echanger( T& a, T& b ) { T c=a; a=b; b=c; }
```

Ici le mot réservé `template` signale qu'il s'agit non d'une fonction, mais d'un *modèle de fonction*, aussi appelé une *fonction générique*. Le mot réservé `typename` précède le type indéterminé, en l'occurrence appelé `T`. Ensuite dans la liste des paramètres et les instructions de la fonctions, on utilise `T` comme si c'était un type usuel. Si vous appelez la fonction `echanger` avec deux paramètres de type `int` le compilateur prendra le modèle ci-dessus et créera aussitôt la fonction concrète `void echanger(int& a,int& b)` comme souhaitée, simplement en remplaçant `T` par `int`. Malin, non ? La fonction concrète ainsi générée est appelée une *instance* du modèle.

Les versions génériques de nos fonctions de tri s'écrivent alors ainsi :

```
template <typename T>
void trier( T& a, T& b ) { if ( a > b ) echanger(a,b); }
```

```
template <typename T>
void trier( T& a, T& b, T& c )
{ if ( a > b ) echanger(a,b);
  if ( b > c ) { echanger(b,c); if ( a > b ) echanger(a,b); }; }
```

Remarque 4.1. Nous avons déjà fait la connaissance de la programmation générique en chapitre I : la classe `vector` est en fait une classe générique dans le sens expliqué plus haut. Sa déclaration est donc

```
template <typename T>
class vector;
```

Si l'on veut l'utiliser pour les éléments de type `int`, par exemple, le compilateur prend le modèle `vector` et en construit la classe souhaitée `vector<int>` simplement en remplaçant `T` par `int`. Avec le même effort de programmation, la classe générique peut ainsi être utilisée dans des situations les plus variées.

Exercice/P 4.2. En s'inspirant de l'opérateur de sortie `<<` défini dans le programme I.18, définir un opérateur de sortie générique qui soit capable d'afficher un vecteur d'un type `T` quelconque. (Pour une solution voir `vectorio.cc`).

4.3. Implémentation de recherche et tri en C++. Si vous voulez, vous pouvez mettre en œuvre les méthodes de recherche et de tri discutées ci-dessus en implémentant une ou plusieurs des fonctions suivantes en C++. Essayez dès le début d´écrire des fonctions génériques comme expliqué en §4. Si pour certaines fonctions vous n'y arrivez pas, vous pouvez avoir recours aux solutions offertes par la bibliothèque STL, discutées dans le paragraphe suivant.

Exercice/P 4.3. Écrire une fonction générique qui teste si le vecteur passé en paramètre est trié :

```
template <typename T>
bool est_trie( const vector<T>& vec );
```

Expliquer pourquoi ce mode de passage est le mieux adapté pour cette tâche.

Exercice/P 4.4. Écrire une fonction générique qui effectue une recherche dichotomique :

```
template <typename T>
int recherche_dichotomique( const vector<T>& vec, const T& element );
```

Expliquer à nouveau pourquoi ce mode de passage est le mieux adapté.

Pour les méthodes de tri vous pouvez choisir entre le tri rapide et le tri fusion :

Exercice/P 4.5. Implémenter le tri rapide en définissant la fonction générique suivante :

```
template <typename T>
void tri_rapide( vector<T>& vec, int gauche, int droite )
```

Ensuite, pour l'appel initial du tri rapide, on pourrait fournir la fonction suivante :

```
template <typename T>
void tri_rapide( vector<T>& vec ) { tri_rapide( vec, 0, vec.size()-1 ); }
```

À noter que les appels récursifs passent le vecteur tout entier comme paramètre par référence ; ce sont les indices `gauche` et `droite` qui spécifie la partie à trier. Expliquer pourquoi ce mode de passage est le mieux adapté pour éviter tout recopiage inutile.

Exercice/P 4.6. Implémenter le tri fusion en définissant la fonction suivante :

```
template <typename T>
void tri_fusion( vector<T>& vec )
```

Contrairement au tri rapide le tri fusion nécessitera des copies dans des vecteurs auxiliaires.
Remarque. — Dans le tri fusion ainsi que le tri rapide il est en général plus efficace de trier les toutes petites listes (disons $n \leq 3$) « à la main » comme en §2.1 Voyez-vous pourquoi ?

Exercice/P 4.7. Vérifiez votre implémentation puis testez sa correction sur beaucoup d'exemples. Si vous voulez tester systématiquement un grand nombre d'exemples, vous pouvez écrire une fonction qui le fait pour vous : d'abord on crée un vecteur aléatoire, ensuite on le trie puis vérifie le résultat par la fonction de l'exercice 4.3. Sur un tel vecteur ordonné on peut finalement tester la recherche dichotomique en cherchant un par un les éléments du vecteur.

Exercice/P 4.8. Mesurer la performance de votre implémentation : combien de temps faut-il pour trier une liste de taille 10^2 ? 10^3 ? 10^4 ? 10^5 ? 10^6 ? Comparer avec une méthode de tri élémentaire. Êtes-vous content ?

5. Solutions fournies par la STL

5.1. Algorithmes de recherche et tri. Comme le tri et la recherche sont omniprésents dans la programmation, la bibliothèque STL en fournit un bon nombre de fonctions génériques. Elles sont disponibles après inclusion du fichier en-tête correspondant par la directive `#include <algorithm>` ; le programme V.1 ci-dessous en illustre l'usage.

Programme V.1 Tri et recherche avec les algorithmes de la STL `tri.cc`

```
#include <iostream>     // pour déclarer l'entrée-sortie standard
#include <algorithm>    // pour définir les fonctions de tri et de recherche
#include <vector>       // pour définir la class générique vector
#include <cstdlib>      // pour déclarer la fonction random()
using namespace std;

template <typename T>
ostream& operator << ( ostream& out, const vector<T>& vec )
{
  out << "( ";
  for ( size_t i=0; i<vec.size(); ++i ) out << vec[i] << " ";
  out << ")";
  return out;
}

void aleatoire( vector<int>& vec  )
{
  for ( size_t i=0; i<vec.size(); ++i ) vec[i]= random() % vec.size();
}

int main()
{
  cout << "\nBienvenue aux vecteurs aléatoires !" << endl;
  cout << "Entrez la longueur svp : ";
  int longueur;
  cin >> longueur;
  vector<int> vec(longueur);
  aleatoire(vec);
  cout << "\nvoici un vecteur aléatoire :  " << vec << endl;
  sort( vec.begin(), vec.end() );
  cout << "\naprès un tri rapide :         " << vec << endl;
  random_shuffle( vec.begin(), vec.end() );
  cout << "\naprès une mélange aléatoire : " << vec << endl;
  stable_sort( vec.begin(), vec.end() );
  cout << "\naprès un tri fusion :         " << vec << endl;
  cout << "\nEntrez un nombre à chercher svp : ";
  int element;
  cin >> element;
  bool trouve= binary_search( vec.begin(), vec.end(), element );
  if ( trouve ) cout << "L'élément " << element << " appartient à la liste.\n";
  else cout << "L'élément " << element << " n'appartient pas à la liste.\n";
  cout << "Au revoir.\n" << endl;
}
```

Pour trier un vecteur `vec`, la STL prévoit entre autre les deux fonctions suivantes :

```
sort( vec.begin(), vec.end() );         // tri rapide (quick sort)
stable_sort( vec.begin(), vec.end() );  // tri fusion (merge sort)
```

La première effectue un tri rapide, la deuxième un tri fusion. Comme expliqué plus haut le tri fusion est légèrement moins rapide et nécessite de la mémoire auxiliaire, mais il *garantit* un temps d'exécution majoré par $\alpha n \log_2 n$. Le tri rapide, par contre, est un peu plus rapide *en moyenne*, mais très lent dans le pire des cas. À choisir en fonction des applications envisagées.

Ajoutons qu'il existe une fonction « inverse » qui mélange un vecteur de manière aléatoire :

```
random_shuffle( vec.begin(), vec.end() );
```

La fonction `binary_search` effectue une recherche dichotomique d'un élément `a` dans un vecteur ordonné :

```
binary_search( vec.begin(), vec.end(), a );
```

Elle renvoie le booléen `true` si `a` est un élément du vecteur, et `false` sinon. Il existe aussi des variantes qui renvoient le premier ou le dernier indice `k` tel que `vec[k]` vaut `a`. Pour en savoir plus sur les algorithmes de la STL, consulter la documentation sur le site `http://www.sgi.com/stl/`.

Le plus simple de tous ces algorithmes est l'échange de deux éléments, déjà rencontré en §4 :

```
template <typename T>
void swap( T& a, T& b ) { T c=a; a=b; b=c; };
```

Notons cependant que la fonction `swap` peut encore être optimisée : si les objects à échanger sont grands, il est plus efficace d'échanger des pointer au lieu des données elles-mêmes. Pour cette raison la STL implémente des fonctions `swap` spécialisée et optimisée sur mesure pour des vecteurs, des listes, des chaînes de catactères, etc. Lorsqu'une version spécialisée est disponible, c'est elle qui sera utilisée. Seulement par défaut utilise-t-on la version générique ci-dessus.

5.2. La classe générique set. La bibliothèque STL fournit une classe générique `set` pour modéliser un *ensemble fini* d'objets d'un type donné. Dans l'exemple qui suit on regarde une variable `ensemble` du type `set<int>`, mais tout s'adapte à n'importe quel autre type au lieu de `int`. Voici les principales opérations que l'on veut effectuer sur un tel ensemble :

(1) Ajouter un élément `a` : ceci se réalise par la fonction `ensemble.insert(a)`.

(2) Enlever un élément `a` : ceci se réalise par la fonction `ensemble.erase(a)`.

(3) Tester si `a` est un élément de `ensemble` : ceci peut se réaliser par `ensemble.count(a)`.

Remarque 5.1. On voit déjà que l'implémentation des ensembles est étroitement liée aux questions de tri et de recherche. Bien sûr on peut implémenter un ensemble comme une liste non ordonnée de n éléments. Pourtant, ceci entraînerait que le test « `a` appartient à `ensemble` » doit parcourir toute la liste, ce qui nécessite un temps proportionnel à n (voir §1). Pour des applications réalistes cette méthode n'est pas praticable.

Dans le souci d'efficacité il est donc nécessaire de stocker les éléments de manière ordonnée. Dans ce cas le test d'appartenance se réalise en temps $\alpha \log_2 n$, par une recherche dichotomique (voir §1). Pour que les opérations `insert` et `erase` soient aussi efficaces, on stocke les éléments non dans un vecteur mais dans une structure mieux adaptée : un arbre binaire.

En faisant abstraction des détails d'une telle implémentation, la conclusion est la suivante :

Théorème 5.2. *On peut implémenter un ensemble fini de sorte que les opérations élémentaires ci-dessus s'effectuent en un temps majoré par $\alpha \log_2 n$, où n est la taille de l'ensemble.* □

Soulignons à nouveau l'importance d'une implémentation efficace : Pour traiter un ensemble d'un million d'éléments, disons, on préfère attendre quelques secondes plutôt que quelques heures !

Le programme V.2 illustre l'utilisation de la classe `set`. Comme vous constaterez, les éléments d'un ensemble sont toujours stockés de manière ordonnée, et chaque élément y apparaît au plus une fois (ce qui sont les principales différences entre un ensemble et une liste).

Pour modéliser des familles avec répétitions possibles on utilise la classe générique `multiset`. Remplacer `set` par `multiset` dans le programme V.2, le tester puis expliquer les différences dans le comportement du logiciel. Le souci d'harmoniser les fonctions pour `set` et `multiset` explique aussi le nom, a priori bizarre, de la fonction `set.count(a)`.

5.3. Les itérateurs. Il y a un concept annexe qui est d'une grande importance pratique : pour afficher un ensemble on souhaite le parcourir du début à la fin. Comme indiqué plus haut, les éléments ne sont pas stockés dans un vecteur, en particulier l'utilisateur ne peut pas les adresser par indice. La solution : on parcourt un ensemble à l'aide d'un *itérateur*. Ceci est illustré dans le programme V.2 : l'itérateur `it` est

Programme V.2 Ensemble d'entiers modélisé par `set<int>` `set.cc`

```
1   #include <iostream>     // déclarer l'entrée-sortie standard
2   #include <set>          // pour définir la class générique set
3   using namespace std;
4
5   template <typename T>
6   ostream& operator << ( ostream& out, const set<T>& ens )
7   {
8     out << "{ ";
9     typename set<T>::const_iterator it;
10    for ( it= ens.begin(); it!=ens.end(); ++it )
11      out << *it << " ";
12    out << "}";
13    return out;
14  }
15
16  int main()
17  {
18    cout << "\nBienvenue aux ensembles ! On teste ici la classe set.\n"
19         << "Entrez un entier positif pour l'ajouter, négatif pour effacer, "
20         << "0 pour terminer." << endl;
21    set<int> ensemble;
22    for(;;)
23      {
24        cout << "ensemble = " << ensemble << endl;
25        cout << "Entrez un entier svp : ";
26        int element;
27        cin >> element;
28        if ( element == 0 ) break;
29        if ( element  > 0 ) ensemble.insert( element );
30        else                ensemble.erase( -element );
31      }
32    for(;;)
33      {
34        cout << "Entrez un nombre à chercher svp : ";
35        int element;
36        cin >> element;
37        if ( element == 0 ) break;
38        if ( ensemble.count( element ) > 0 )
39          cout << "L'élément " << element << " appartient à l'ensemble.\n";
40        else
41          cout << "L'élément " << element << " n'appartient pas à l'ensemble.\n";
42      }
43    cout << "Au revoir.\n" << endl;
44  }
```

une sorte de pointeur sur un élément de notre ensemble. On peut l'incrémenter par `++it` et décrémenter par `--it`, pour avancer ou reculer. Puis on peut accéder à l'élément vers lequel il pointe par `*it`. Pour parcourir du début à la fin, l'ensemble `ens` fournit les itérateurs `ens.begin()` et `ens.end()`. Comme vous pouvez le constater, la construction d'une boucle devient ainsi facile et naturelle.

Mais vérifiez tout : ce qui est bon, retenez-le !
1 Thess. **5** 21

PROJET V

Applications du tri aux équations diophantiennes

Objectif. Le tri peut se révéler un outil magnifique même dans des situations inattendues. Dans ce projet nous allons appliquer le tri aux équations $a^3+b^3=c^3+d^3$ et $a^4+b^4=c^4+d^4$. Finalement on reprend la conjecture d'Euler concernant l'équation $a^4+b^4+c^4=d^4$. Il y aura des surprises…

Ce projet nous permet entre autre de tester nos méthodes de tri et de recherche sur des exemples réalistes, et avec des algorithmes efficaces le temps d'exécution restera raisonnable. On constatera que la mémoire disponible peut également être une ressource précieuse.

Sommaire

1. Le taxi de Ramanujan

L'anecdote suivante est devenue légendaire : Un jour quand le mathématicien G.H. Hardy rendit visite à son collègue et ami S. Ramanujan, il mentionna le numéro du taxi avec lequel il était venu, en ajoutant « *C'est sans doute un nombre sans aucune propriété intéressante.* » « *Au contraire !* » répondit Ramanujan « *C'est le plus petit nombre qui se décompose de deux manières différentes en somme de deux cubes d'entiers positifs.* » Peut-être vous connaissez cette histoire mais vous avez oublié le numéro du taxi. Comment le retrouver rapidement ?

— ∗ —

Nous nous proposons ici de chercher les plus petits nombres qui s'écrivent de deux manières différentes comme somme de deux cubes d'entiers positifs. Autrement dit nous cherchons les petites solutions de l'équation $a^3+b^3=c^3+d^3$ avec $a,b,c,d\in\mathbb{Z}_+$ et $\{a,b\}\neq\{c,d\}$. Voici trois démarches qui se présentent :

Exercice 1.1. On peut effectuer une recherche exhaustive qui parcourt les quadruplets (a,b,c,d) vérifiant $1\leq a<c\leq d<b\leq N$ (le justifier). Vérifier que cette méthode nécessite $\binom{N+1}{4}$ itérations.

Exercice 1.2. En s'inspirant du projet I, expliciter une méthode qui ne nécessite que $\binom{N}{3}$ itérations grâce à une fonction auxiliaire $n\mapsto\lfloor\sqrt[3]{n}\rfloor$. En quoi cette approche est-elle préférable à la méthode précédente ?

Exercice 1.3. Détaillez une méthode utilisant le tri : pour $1\leq a\leq b\leq N$ on construit une liste des valeurs a^3+b^3, on la trie, puis on cherche les doublons. Combien de mémoire est necessaire pour stocker la liste ? Combien d'itérations sont nécessaire pour le tri puis la recherche ?

Remarque 1.4. Comme quatrième approche vous pouvez effectuer une recherche sur internet. Esquisser en quoi cette dernière approche, elle aussi, est basée sur des méthodes de tri et de recherche.

Exercice/P 1.5. Choisissez la méthode qui vous semble la plus efficace et/ou la plus simple à réaliser, puis justifier votre choix. Implémentez-la afin de trouver le numéro du taxi de Ramanujan. Ce résultat serait-il également accessible par les autres méthodes ? Combien de solutions y a-t-il pour $N=500$? $N=1000$? $N=5000$? Jusqu'où peut-on aller ?

Exercice 1.6. Trouver le plus petit nombre qui s'écrive de *trois* manières différentes comme somme de deux cubes d'entiers positifs. (À l'heure actuelle on connaît la réponse pour les multiplicités 2, 3, 4, et 5 seulement.)

Exercice 1.7. Pouvez-vous trouver le plus petit nombre qui s'écrive de deux manières différentes comme somme de deux bicarrés d'entiers positifs, $a^4+b^4=c^4+d^4$? (Dans ce cas on ne connaît aucun exemple de multiplicité ≥ 3.)

1.1. Équations diophantiennes, recherche et tri. En généralisant l'exemple précédent on arrive à la formulation suivante : étant données deux fonctions $P,Q\colon \mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}$, comment énumérer rapidement toutes les solutions de l'équation $P(a,b)=Q(c,d)$ avec $(a,b,c,d)\in[\![1,N]\!]^4$? Bien sûr on peut parcourir tous les quadruplets (a,b,c,d) et tester si l'équation $P(a,b)=Q(c,d)$ est satisfaite. Ceci peut s'implémenter par quatre boucles imbriquées, et nécessite N^4 itérations au total. Peut-on faire mieux ?

Exercice 1.8. Expliciter une méthode bénéficiant d'une méthode efficace de tri :

(1) On dresse une liste de tous les triplets $(P(a,b),a,b)$, puis on la trie par rapport à P.

(2) On dresse une liste de tous les triplets $(Q(c,d),c,d)$, puis on la trie par rapport à Q.

(3) On parcourt les listes simultanément en cherchant les coïncidences $P(a,b)=Q(c,d)$.

Estimer la mémoire nécessaire et le temps d'exécution. Pourquoi est-il hors de question d'utiliser un tri de complexité quadratique ici ?

Exercice 1.9. Expliciter une variante bénéficiant des méthodes efficaces de tri et de recherche :

(1) On dresse une liste de tous les triplets $(Q(c,d),c,d)$, puis on la trie par rapport à Q.

(2) On parcourt tous les triplets $(P(a,b),a,b)$ en cherchant les coïncidences avec la liste.

Estimer la mémoire nécessaire et le temps d'exécution. Pourquoi est-il hors de question d'utiliser une recherche linéaire ici ?

2. L'équation $a^4+b^4+c^4=d^4$ reconsidérée

Regardons à nouveau l'équation $a^4+b^4+c^4=d^4$ vue en projet I. L'approche ci-dessus s'applique avec $P(a,b)=a^4+b^4$ et $Q(c,d)=d^4-c^4$. Nos nouvelles méthodes sont plus efficaces que l'approche du projet I et nous permettrons de chercher des solutions dans un domaine beaucoup plus large ! Le but ultime sera de trouver toutes les solutions $(a,b,c,d)\in[\![1,N]\!]^4$ avec $N=10^6$. Comme dans le projet I nous posons

```
typedef unsigned long long int Integer;
```

Exercice/P 2.1. Chercher les solutions jusqu'à $N=2000$ par la méthode esquissée ci-dessus : stocker les valeurs d^4-c^4 avec $1\leq c<d\leq N$ dans un vecteur de type `vector<Integer>`, le trier, puis chercher les valeurs a^4+b^4 avec $1\leq a\leq b\leq N$ dans la liste.

☞ *Étapes intermédiaires :* Comme d'habitude il sera utile que le programme affiche les différentes étapes (construction, tri, recherche) et l'avancement du travail (pour les boucles extérieures). Si vous avez bien programmé, votre logiciel s'exécute en quelques secondes. Félicitations !

Remarque 2.2. Pour $N=2000$ il faut construire un vecteur de longueur $\frac{1}{2}N(N-1)=1999000$. Comme on connaît sa taille d'avance, il est utile de *réserver* la mémoire tout au début, par l'instruction suivante :

```
vector<Integer> imageQ; imageQ.reserve(max*(max-1)/2);
```

Ceci définit un vecteur vide mais prévoit déjà la mémoire indiquée. Ensuite vous pouvez remplir le vecteur par la fonction `imageQ.push_back()`, tout en évitant la réallocation inutile pendant la construction.

Remarque 2.3. Notre approche marche encore pour $N=5000$, peut-être même jusqu'à $N=10000$, mais au delà les limites de cette méthode se font sentir : le tester. il se trouve qu'ici la mémoire devient le facteur limitant. De combien de mémoire vive dispose la machine sur laquelle vous travaillez ? Quand la mémoire commence à manquer il se peut que le tri fusion n'aboutit plus alors que le tri rapide marche encore (expliquer pourquoi).

2.1. Restrictions modulaires. Jusqu'ici notre méthode est très générale est s'applique à des fonctions $P,Q\colon\mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}$ quelconques. Pour aller plus loin nous utilisons le fait que $P(a,b)=a^4+b^4$ et $Q(c,d)=d^4-c^4$ sont des *polynômes* et de plus *homogènes*.

Exercice/M 2.4. Vérifier que toute solution $(a,b,c,d)\in\mathbb{Z}^4$ de l'équation $P(a,b)=Q(c,d)$ entraîne une famille infinie de solutions (ka,kb,kc,kd) avec $k\in\mathbb{Z}$. Justifier que notre recherche peut se restreindre aux solutions *primitives*, c'est-à-dire celles qui vérifient $\operatorname{pgcd}(a,b,c,d)=1$.

Exercice/M 2.5. Vérifier que toute solution $(a,b,c,d)\in\mathbb{Z}^4$ de l'équation $P(a,b)=Q(c,d)$ induit une solution $(\overline{a},\overline{b},\overline{c},\overline{d})\in\mathbb{Z}_n^4$ de l'équation $\overline{P}(\overline{a},\overline{b})=\overline{Q}(\overline{c},\overline{d})$ modulo n. Par contraposé, comment l'inégalité $\overline{P}(\overline{a},\overline{b})\neq\overline{Q}(\overline{c},\overline{d})$ peut-elle simplifier la recherche des solutions dans $\mathbb{Z}^4$?

Exercice/M 2.6. Montrer que l'application $\mathbb{Z}_4 \to \mathbb{Z}_4$, $x \mapsto x^4$ a pour l'image l'ensemble $\{0,1\}$. En déduire que, quitte à permuter a,b,c, toute solution $(a,b,c,d) \in \mathbb{Z}^4$ vérifie $a \equiv b \equiv 0 \pmod 2$. Pour une solution primitive on peut de plus conclure que $c \equiv d \equiv 1 \pmod 2$.

Exercice/M 2.7. Montrer que l'application $\mathbb{Z}_5 \to \mathbb{Z}_5$, $x \mapsto x^4$ a pour l'image l'ensemble $\{0,1\}$. Quitte à permuter a,b,c, toute solution $(a,b,c,d) \in \mathbb{Z}^4$ vérifie donc $a \equiv b \equiv 0 \pmod 5$. Pour une solution primitive on peut de plus conclure que $c \not\equiv 0 \not\equiv d \pmod 5$.

Avec cette technique on arrive au résultat suivant, que l'on admettra (cf. M. Ward, *Euler's problem on sums of three fourth powers*, Duke Mathematical Journal 15 (1948) 827–837) :

Théorème 2.8 (dû à M. Ward, 1948)**.** *Si $(a,b,c,d) \in \mathbb{Z}^4$ est une solution primitive de l'équation $a^4+b^4+c^4=d^4$, alors $d \not\equiv 0 \pmod 5$ et $d \equiv 1 \pmod 8$. Quitte à permuter les variables a,b,c on a $a \equiv 0 \pmod 8$ et $b \equiv 0 \pmod{40}$ ainsi que $c \equiv \pm d \pmod{1024}$.* □

☞ *Optimisation :* L'intérêt pratique de ce théorème est qu'il suffit désormais de parcourir les couples (a,b) et (c,d) vérifiant les dites congruences. En C++ ceci peut se réaliser par les boucles suivantes :

```
for ( Integer a=8; a<=max; a+=8 )
  for ( Integer b=40; b<=max; b+=40 ) { ... };
for ( Integer d=1; d<=max; d+=8 ) if ( d%5 != 0 )
  { for ( Integer c=d%1024; c<d; c+=1024 ) { ... };
    for ( Integer c=1024-d%1024; c<d; c+=1024 ) { ... }; };
```

Cette astuce écarte plus de 99% de couples $(a,b) \in [\![1,N]\!]^2$ car on n'en retient qu'une fraction $\frac{1}{320}$; mieux encore, il écarte 99,99% des couples $(c,d) \in [\![1,N]\!]^2$ car on n'en retient qu'une fraction $\frac{1}{10240}$ (le justifier).

Exercice/P 2.9. Chercher toutes les solutions jusqu'à $N = 100000$ par la méthode améliorée. Vérifier que l'on construit un vecteur de longueur 971529 seulement.

Si vous voulez aller encore un peu plus loin, vous pouvez écarter 99% des couples (c,d) restants en appliquant d'autres cribles modulaires : on fixe un nombre n et on regarde quels couples (c,d) apparaissent dans les solutions de l'équation $\overline{a}^4+\overline{b}^4=\overline{d}^4-\overline{c}^4$ modulo n. Si (c,d) est impossible modulo n, alors il est impossible dans $\mathbb{Z}$. Le programme V.3 montre une implémentation de cette idée en C++.

Exercice/P 2.10. Chercher toutes les solutions jusqu'à $N = 500000$ par la méthode améliorée. Vérifier que l'on construit un vecteur de longueur 418127 seulement.

☞ *Avertissement.* — Le choix du type `long long int` entraîne que nous travaillons modulo 2^{64} : si le logiciel trouve une solution, ceci ne veut dire que $a^4+b^4+c^4 \equiv d^4 \pmod{2^{64}}$. Interprété correctement, c'est un résultat précieux, car une fois trouvée, il est très facile de confirmer ou réfuter une solution :

Pour une solution prospective a,b donnée, chercher les valeurs c,d associéés, puis afficher le quadruplet (a,b,c,d) comme « candidat ». Ensuite on peut effectuer un calcul dans $\mathbb{Z}$ pour vérifier si (a,b,c,d) est effectivement une solution. (On pourra utiliser `mpz_class` pour le calcul avec des grands entiers.)

Remarque 2.11. Pour résumer, nous avons raffiné successivement nos méthodes : l'idée de la méthode 1, notre point de départ, est de stocker les valeurs $P(c,d)$ dans un vecteur trié. La méthode 2 restreint les couples (c,d) par certaines congruences (théorème 2.8). La méthode 3 généralise cette idée et implémente plusieurs cribles modulaires (programme V.3). Pour souligner l'utilité de ces raffinements successifs, le tableau ci-dessus présente le nombre des couples (c,d) à considérer avec $1 \le c < d \le N$. Grâce aux restrictions modulaires exploitées par la méthode 3, la construction du vecteur se fait assez rapidement, et le tri suivant se passe également sans problème. C'est la dernière phase, la recherche des coïncidences, qui est la plus coûteuse en temps. Pour $N = 10^6$ il faut compter environ une heure d'exécution.

$N=$	5000	10000	50000	100000	500000	1000000
méthode 1	12497500	49995000	1249975000	4999950000	124999750000	499999500000
méthode 2	2194	9268	241627	971529	24388874	97605862
méthode 3	61	183	3623	15373	418127	1672259

Remarque 2.12. On peut encore optimiser la mémoire nécessaire pour l'énumération exhaustive, en utilisant une *file de priorité* au lieu d'un tableau. Pour en savoir plus, consultez l'article de Daniel J. Bernstein, *Enumerating solutions to $p(a)+q(b)=r(c)+s(d)$*, Mathematics of Computation 70 (2001), 389–394.

Programme V.3 Cribles modulaires pour le problème d'Euler `crible.cc`

```
#include <iostream>
#include <vector>
using namespace std;
typedef unsigned long long int Integer;

// Calculer  P(a,b) = a^4 + b^4  modulo mod (sans débordement)
Integer lePolynomeP( Integer a, Integer b, Integer mod )
{
  a%=mod; a*=a; a%=mod; a*=a; a%=mod;
  b%=mod; b*=b; b%=mod; b*=b; b%=mod;
  return (a+b) % mod;
}

// Calculer  Q(c,d) = d^4 - c^4  modulo mod (sans débordement)
Integer lePolynomeQ( Integer c, Integer d, Integer mod )
{
  c%=mod; c*=c; c%=mod; c*=c; c%=mod;
  d%=mod; d*=d; d%=mod; d*=d; d%=mod;
  return ( c<=d ? d-c : d-c+mod );
}

// Pour tenir compte des restrictions modulaires, la fonction suivante
// construit un vecteur  image  du type  vector<bool>  et de longueur mod
// de sorte que  image[x]==true  ssi x est dans l'image de P modulo mod.
vector<bool> imagePmod( const int& mod )
{
  cout << "construction de l'image de P mod " << mod << " ... " << flush;
  vector<bool> image(mod,false);
  for ( int a=0; a<mod; ++a )
    for ( int b=0; b<mod; ++b )
      image[ lePolynomeP(a,b,mod) ]= true;
  cout << "statistique : " << flush;
  Integer possibles=0;
  for ( int c=0; c<mod; ++c )
    for ( int d=0; d<mod; ++d )
      if ( image[ lePolynomeQ(c,d,mod) ] ) ++possibles;
  cout << (100*possibles)/(mod*mod) << "% retenus" << endl;
  return image;
}

// Construction des images modulo 5^4,13^2,3^4,29^2,7^3,11^2,19^2,31^2.
// (Dans l'ordre d'efficacité, d'après de nombreux tests empiriques.)
// Les images sont précalculées comme des vecteurs constants,
// ce qui permettra d'accéder rapidement à ces informations.
const vector<bool>
  image625= imagePmod(625), image169= imagePmod(169),
  image81 = imagePmod(81),  image841= imagePmod(841),
  image343= imagePmod(343), image121= imagePmod(121),
  image361= imagePmod(361), image961= imagePmod(961);

// La fonction  crible(c,d)  teste si le couple (c,d) passe les cribles.
// Si non, elle renvoie false et on peut supprimer (c,d) dans la suite.
bool crible( Integer c, Integer d )
{
  return image625[lePolynomeQ(c,d,625)] && image169[lePolynomeQ(c,d,169)]
      && image81 [lePolynomeQ(c,d,81) ] && image841[lePolynomeQ(c,d,841)]
      && image343[lePolynomeQ(c,d,343)] && image121[lePolynomeQ(c,d,121)]
      && image361[lePolynomeQ(c,d,361)] && image961[lePolynomeQ(c,d,961)];
}
```