Données du problème

Un programme informatique peut-être conduit à traiter un grand nombre d'informations. Imaginez un programme de gestion qui traite les feuilles de paye d'une grande entreprise, ou celui du centre des impôts qui gèrent des millions de contribuables ou bien un jeu de simulation où le réalisme d'une scène est directement lié au nombre de facettes représentant les objets de cette scène. Pour prendre un autre exemple, dans le cas des modèles de simulation atmosphérique pour faire des prévisions météorologiques, l'amélioration des prévisions passe, entre autres choses, par l'augmentation de la finesse du maillage du modèle atmosphérique. Le maillage est composé de milliers de points et, pour chaque point, il faut disposer d'une dizaine d'informations comme la température, la pression, la vitesse et la direction du vent,...

Pour traiter plus de données, on peut facilement imaginer qu'en augmentant la puissance de calcul des machines (en rapidité d'exécution ou en taille de données manipulables), on va pouvoir traiter plus d'informations pendant un temps donné. Une autre solution passe par la répartition du calcul sur plusieurs machines lorsque cela est possible : on parle alors de parallélisme. On peut aussi se poser la question de savoir s'il n'est pas possible de réécrire le programme pour qu'il soit plus efficace, l'efficacité d'un programme pouvant se mesurer par la possibilité de traiter plus d'informations dans un temps donné à puissance machine égale.

Pour illustrer mon propos, imaginez que vous disposiez d'un livre dans lequel seraient rangés dans un ordre arbitraire tous les mots de la langue française. Le seul moyen que vous auriez pour trouver un mot dans ce livre serait de parcourir tous les mots les uns après les autres jusqu'à trouver le bon. Cette méthode serait largement fastidieuse pour trouver la définition d'un mot. Il a donc été inventé une représentation astucieuse des données grâce à l'ordre lexicographique qui permet à chaque mot d'avoir une place précise dans le dictionnaire. La représentation des données ne suffit pas car, même avec un dictionnaire, vous pouvez appliquer la même méthode fastidieuse décrite précédemment pour trouver le mot cherché. Dès l'école primaire, on vous apprend un algorithme efficace pour trouver votre mot dans le dictionnaire sans en avoir à tourner toutes les pages et parcourir toutes les colonnes de votre dictionnaire. Dans le cas du dictionnaire, on a à la fois une bonne présentation des données et un algorithme efficace pour exploiter ces données.

La comparaison de l'efficacité de deux programmes informatiques est une chose aisée. Il suffit d'avoir un ordinateur et un chronomètre. On fait exécuter à chacun des programmes la même tâche avec les mêmes données et l'on mesure le temps mis par chacun des programmes, le meilleur étant celui qui a mis le moins de temps. L'inconvénient de la méthode est qu'elle oblige à écrire les deux programmes ce qui ne pose pas vraiment de problèmes quand les programmes ne font qu'une dizaine de lignes, mais ce qui n'est pas économiquement viable si les programmes représentant des milliers de lignes de codes. On aimerait plutôt disposer d'un moyen permettant de dire à partir de la description des algorithmes associés à ces deux programmes lequel sera a priori le plus efficace.

Généralement, on prend une ou plusieurs opérations élémentaires de référence, coûteuses en temps d'exécution et l'on essaie d'évaluer le nombre d'appels à ces opérations en fonction du nombre n de données du problème. Une opération élémentaire pourra être le calcul d'un produit ou bien le calcul d'une racine carrée ou bien l'écriture d'une donnée sur le disque dur de la machine,...

La **complexité** d'un algorithme donne une information sur le comportement de l'algorithme vis-à-vis des données. Si on appelle $T(n)$ le temps d'exécution en fonction de n , on dira qu'un algorithme est à complexité :

- logarithmique si $T(N) = O(\log n)$
- linéaire si $T(N) = O(n)$
- polynomiale si $T(n) = O(n^k)$ avec $k \geq 2$
- exponentielle si $T(n) = O(\lambda^k)$ avec $\lambda > 1$

cas école :

Problème : trouver tous les nombres entre 0 et n qui s'écrivent comme la somme de deux carrés;

```

1ier algorithme : pour i variant de 0 à n faire
    ok reçoit faux;
    pour j variant de 0 à n faire
        pour k variant de 0 à n faire
            si  $i = j^2 + k^2$  alors ok reçoit vrai fin si
        fin pour
    fin pour
    si ok = vrai alors afficher i
fin pour

```

Pour estimer la complexité de l'algorithme précédent ainsi que les suivants, nous allons évaluer le nombre d'additions, de multiplications et d'extraction de racines carrées en fonction de n . Ici le calcul est simple, nous avons trois boucles imbriquées, à chaque passage dans la boucle la plus interne, nous effectuons 1 addition et 2 multiplications, ce qui nous donne pour cet algorithme :

- nombre d'additions : $(n + 1)^3 \sim n^3$
- nombre de multiplications : $2(n + 1)^3 \sim 2n^3$

On peut améliorer l'algorithme précédent, on peut pour chaque i , on peut s'arrêter dès que l'on a trouvé une décomposition de i en somme de 2 carrés.

```

2ième algorithme : pour i variant de 0 à n faire
    ok reçoit faux;
    pour j variant de 0 à n tant que ok = faux faire
        pour k variant de 0 à n tant que ok = faux faire
            si  $i = j^2 + k^2$  alors afficher i; ok reçoit vrai
        fin pour
    fin pour
fin pour

```

Ici on ne peut pas déterminer de manière exacte le nombre d'opérations effectuées, on ne peut obtenir qu'une majoration

- nombre d'additions : inférieur à $(n + 1)^3 \sim n^3$
- nombre de multiplications : inférieur à $2(n + 1)^3 \sim 2n^3$

On peut améliorer l'algorithme précédent en remarquant qu'il n'y a pas besoin pour un i donné de faire varier j de 1 à n mais qu'il suffit de faire varier i de 1 à i et même mieux de 1 à $\sqrt{i}$. Puis pour un couple (i, j) donné de faire varier k de 0 à $\sqrt{i - j^2}$.

3^{ième} algorithme: pour i variant de 0 à n faire

```

ok reçoit faux;
  pour  $j$  variant de 0 à  $\sqrt{i}$  tant que ok = faux faire
    pour  $k$  variant de 0 à  $\sqrt{i - j^2}$  tant que ok = faux faire
      si  $i = j^2 + k^2$  alors afficher  $i$ ; ok reçoit vrai
    fin si
  fin pour
fin pour

```

Ici on peut évaluer les opérations, de la manière suivante

- nombre d'additions : inférieur à $\sum_{i=0}^n \left(1 + \sum_{j=0}^{\sqrt{i}} (1 + \sqrt{i - j^2}) \right) \sim \left(\frac{\pi}{8} n^2 \right)^{(1)}$
- nombre de multiplications : inférieur à $\sum_{i=0}^n \left(1 + 2 \sum_{j=0}^{\sqrt{i}} (1 + \sqrt{i - j^2}) \right) \sim \left(\frac{\pi}{4} n^2 \right)$
- nombre de racines carrées : $\sum_{i=0}^n \sqrt{i} \sim \frac{2}{3} n \sqrt{n}$

On peut encore améliorer notre programme en supprimant la boucle la plus interne. En effet, si on connaît i et que l'on a choisi j , le seul k qui puisse convenir est $\sqrt{i - j^2}$

4^{ième} algorithme: pour i variant de 0 à n faire

```

ok reçoit faux;
  pour  $j$  variant de 0 à  $\sqrt{i}$  tant que ok = faux faire
    si  $(i - j^2)$  est un carré alors afficher  $i$ ; ok reçoit vrai
  fin si
fin pour

```

Ici on peut évaluer les opérations, de la manière suivante

- nombre d'additions : inférieur à $\sum_{i=0}^n \sqrt{i} \sim \frac{2}{3} n \sqrt{n}$

1. L'équivalent a été calculé à l'aide Maple.

- nombre de multiplications : inférieur à $\sum_{i=0}^n \sqrt{i} \sim \frac{2}{3}n\sqrt{n}$
- nombre de racines carrées : $n + 1$

Une dernière amélioration peut être trouvée en prenant un peu de recul par rapport à la question posée et en reformulant cette question. Ce que nous cherchons finalement se sont tous les nombres de la forme $i^2 + j^2$ qui sont plus petits que n et ces nombres sont facilement calculables grâce à deux boucles imbriquées comme cela est décrit dans l'algorithme suivant :

5^{ième} algorithme : pour i variant de 0 à $\sqrt{n}$,
 pour j variant de 0 à $\sqrt{n - i^2}$
 afficher $i^2 + j^2$
 fin pour
 fin pour

Ici on peut évaluer les opérations, de la manière suivante :

- nombre d'additions : $\sum_{i=0}^{\sqrt{n}} (2 + \sqrt{n - i^2}) \sim \frac{\pi}{4}n$
- nombre de multiplications : $\sum_{i=0}^{\sqrt{n}} (3 + 2\sqrt{n - i^2}) \sim \frac{\pi}{2}n$
- nombre de racines carrées : $\sqrt{n} + 2$

Tableau des complexités des algorithmes précédents

	algo n°1	algo n°2	algo n°3	algo n°4	algo n°5
nb de produits	$\sim 2n^3$	$O(2n^3)$	$O\left(\frac{\pi}{4}n^2\right)$	$\sim \left(\frac{2}{3}n\sqrt{n}\right)$	$\sim \frac{\pi}{2}n$
nb de sommes	$\sim n^3$	$O(n^3)$	$O\left(\frac{\pi}{8}n^2\right)$	$\sim \left(\frac{2}{3}n\sqrt{n}\right)$	$\sim \frac{\pi}{4}n$
nb de racines carrés	0	0	$\sim \frac{2}{3}n\sqrt{n}$	n	$\sim \sqrt{n}$

Voici les versions Maple des 5 algorithmes précédents :

```
algo1 := proc(n)  
  local l, i, j, k, ok;  
  l := {};  
  for i from 0 to n do  
    ok := faux;  
    for j to n do for k from 0 to n do if  $i = j^2 + k^2$  then ok := vrai fi od od;  
    if ok = vrai then l := l union {i} fi  
  od;  
  RETURN(l)  
end
```

```
algo2 := proc(n)  
  local l, i, j, k, ok;  
  l := {};  
  for i from 0 to n do  
    ok := faux;  
    for j to n while ok = faux do  
      for k from 0 to n while ok = faux do if  $i = j^2 + k^2$  then ok := vrai fi od  
    od;  
    if ok = vrai then l := l union {i} fi  
  od;  
  RETURN(l)  
end
```

```

algo3 := proc(n)
  local l, i, j, k, ok;
  l := {};
  for i from 0 to n do
    ok := faux;
    for j to evalf(sqrt(i)) while ok = faux do for k from 0 to evalf(sqrt(i - j2)) while
      ok = faux do if i = j2 + k2 then ok := vrai fi
    od
  od;
  if ok = vrai then l := l union {i} fi
od;
  RETURN(l)
end

```

```

algo4 := proc(n)
  local l, i, j, k, ok;
  l := {};
  for i from 0 to n do
    ok := faux;
    for j to evalf(sqrt(i)) while ok = faux do if type(i - j2, square) then ok := vrai fi od;
    if ok = vrai then l := l union {i} fi
  od;
  RETURN(l)
end

```

```

algo5 := proc(n)
  local i, j, l;
  l := {};
  for i from 0 to evalf(sqrt(n)) do
    for j from 0 to evalf(sqrt(n - i2)) do l := l union {i2 + j2} od
  od;
  RETURN(l)
end

```

Tableau de comparaison des temps de calculs des algorithmes précédents pour différentes valeurs de n

n	algo n°1	algo n°2	algo n°3	algo n°4	algo n°5
10	0,151	0,069	0,042	0,192	0,086
20	1,593	1,002	0,096	0,069	0,046
50	22,467	13,759	0,411	0,260	0,069
100	180,730	118,875	1,716	1,179	0,077
1000			116,646	27,042	0,605
10000					17,644

Exercice : Donner un algorithme retournant les nombres compris entre 1 et n qui ne se décomposent pas comme la somme de deux nombres premiers. On pourra utiliser les fonctions Maple suivantes :

ithprime(p) : qui retourne le $p^{i\grave{e}me}$ nombre premier,

isprime(p) : qui retourne *true* si p est un nombre premier et *false* sinon.