Préparation à l'oral du CAPES de mathématiques

Exemples d'algorithmes

Principes, preuves, complexité et applications pédagogiques

Table des matières

1 Prérequis et objectifs **2**

2 Qu'est-ce qu'un algorithme ? **2**
2.1 Les principales structures . . . 2
2.2 Correction et terminaison . . . 2
2.3 Complexité . . . 3

3 Algorithme 1 : calcul du PGCD **3**
3.1 Problème . . . 3
3.2 Algorithme d'Euclide . . . 3
3.3 Exemple d'exécution . . . 3
3.4 Preuve de correction . . . 3
3.5 Prolongement : algorithme d'Euclide étendu . . . 4

4 Algorithme 2 : recherche dichotomique **4**
4.1 Problème . . . 4
4.2 Invariant de boucle . . . 4
4.3 Complexité . . . 4

5 Algorithme 3 : calcul d'une somme **5**
5.1 Somme des premiers entiers . . . 5
5.2 Justification par invariant . . . 5
5.3 Deux méthodes à comparer . . . 5

6 Algorithme 4 : méthode de dichotomie pour une équation **5**
6.1 Principe . . . 5
6.2 Erreur et nombre d'itérations . . . 6

7 Algorithme 5 : approximation de la racine carrée **6**
7.1 Méthode de Héron . . . 6
7.2 Propriétés . . . 7

8 Algorithme 6 : tri par insertion **7**
8.1 Principe . . . 7
8.2 Correction . . . 7
8.3 Complexité . . . 8

9 Algorithme 7 : calcul rapide d'une puissance **8**
9.1 Exponentiation rapide . . . 8
9.2 Correction et complexité . . . 8

10 Comparaison des algorithmes **9**

11 Démarche de résolution d'un problème algorithmique **9**

12 Exploitation pédagogique **9**

13 Liste d'exercices **10**

14 Questions possibles du jury **12**

15 Conclusion **12**

1 Prérequis et objectifs

Prérequis

On suppose connus les notions suivantes :

— calcul numérique et littéral ;

— raisonnement logique, implication et récurrence ;

— suites numériques et fonctions ;

— notions élémentaires de divisibilité et d'arithmétique ;

— lecture d'un programme comportant des variables, des affectations, des tests et des boucles.

Objectifs

Cette leçon vise à montrer comment un problème mathématique peut être résolu par une suite finie d'instructions précises. Il s'agit de savoir concevoir un algorithme, justifier sa correction, étudier son arrêt et comparer plusieurs méthodes selon leur efficacité.

Les exemples choisis couvrent l'arithmétique, le calcul numérique, la recherche dans une liste et le tri. Ils permettent d'aborder les notions de boucle, de récursion, d'invariant, de terminaison, de complexité et de validation expérimentale.

L'enjeu pédagogique est également de relier l'algorithmique à l'activité mathématique : conjecturer, expérimenter, démontrer et interpréter un résultat.

2 Qu'est-ce qu'un algorithme ?

Définition 2.1. *Un algorithme est une méthode de résolution d'un problème, décrite par une suite finie d'instructions non ambiguës, applicables à des données d'entrée et produisant un résultat.*

Un algorithme doit être distingué de son implémentation : le premier décrit une méthode, tandis que le programme traduit cette méthode dans un langage donné.

2.1 Les principales structures

— **Séquence :** exécuter des instructions dans un ordre déterminé.

— **Condition :** choisir une branche selon un test.

— **Itération :** répéter des instructions tant qu'une condition est vérifiée ou pendant un nombre donné d'étapes.

— **Récursion :** résoudre un problème en faisant appel à une version plus simple du même problème.

2.2 Correction et terminaison

La correction d'un algorithme comporte deux aspects :

— la *correction partielle* : si l'algorithme termine, son résultat satisfait la spécification ;

— la *terminaison* : l'algorithme finit après un nombre fini d'étapes pour toutes les entrées autorisées.

La correction totale réunit ces deux propriétés.

2.3 Complexité

On étudie notamment :

— la complexité temporelle : nombre d'opérations élémentaires en fonction de la taille des données ;

— la complexité spatiale : quantité de mémoire supplémentaire utilisée.

La notation $\mathcal{O}(n)$ décrit un ordre de grandeur asymptotique, à un facteur multiplicatif près, lorsque n devient grand. Elle ne donne pas directement un temps d'exécution en secondes.

3 Algorithme 1 : calcul du PGCD

3.1 Problème

Étant donnés deux entiers naturels a et b, non tous deux nuls, déterminer leur plus grand diviseur commun.

3.2 Algorithme d'Euclide

L'identité fondamentale est

$$\mathrm{PGCD}(a, b) = \mathrm{PGCD}(b, a \bmod b), \qquad b > 0.$$

Elle permet de remplacer un couple d'entiers par un couple plus simple sans modifier son PGCD.

```
def pgcd(a, b):
    a = abs(a)
    b = abs(b)

    while b != 0:
        a, b = b, a % b

    return a
```

Listing 1 – Algorithme d'Euclide en Python

3.3 Exemple d'exécution

Calculons le PGCD de 252 et 105 :

$$\begin{aligned} 252 &= 2 \times 105 + 42, \\ 105 &= 2 \times 42 + 21, \\ 42 &= 2 \times 21 + 0. \end{aligned}$$

Ainsi,

$$\mathrm{PGCD}(252, 105) = 21.$$

3.4 Preuve de correction

À chaque étape, l'identité d'Euclide garantit que le PGCD ne change pas. Lorsque le second terme devient nul, le premier terme est le PGCD recherché.

La terminaison découle du fait que le second terme est un entier naturel strictement décroissant tant qu'il est non nul.

Pour des entiers de taille comparable à n, l'algorithme d'Euclide effectue un nombre d'étapes logarithmique en la taille des entiers, dans le cas usuel de l'analyse binaire des entrées.

3.5 Prolongement : algorithme d'Euclide étendu

On peut calculer simultanément des entiers u et v tels que

$$au + bv = \mathrm{PGCD}(a, b).$$

Cette identité de Bézout intervient notamment dans la résolution d'équations diophantiennes et dans l'arithmétique modulaire.

4 Algorithme 2 : recherche dichotomique

4.1 Problème

Rechercher la présence d'une valeur x dans une liste triée de n éléments.

La dichotomie exploite le fait que les données sont ordonnées. À chaque étape, on compare x à l'élément central et l'on élimine au moins la moitié de la zone de recherche.

```
def recherche_dichotomique(T, x):
    gauche = 0
    droite = len(T) - 1

    while gauche <= droite:
        milieu = (gauche + droite) // 2

        if T[milieu] == x:
            return milieu
        elif T[milieu] < x:
            gauche = milieu + 1
        else:
            droite = milieu - 1

    return -1
```

Listing 2 – Recherche dichotomique

4.2 Invariant de boucle

Si x est présent dans le tableau et n'a pas encore été trouvé, une occurrence de x appartient à l'intervalle d'indices [`gauche`, `droite`].

L'initialisation est immédiate. À chaque comparaison, l'ordre du tableau justifie l'élimination de la moitié impossible. L'invariant est donc conservé.

Lorsque l'intervalle devient vide, aucune occurrence ne peut subsister dans la zone initialement recherchée.

4.3 Complexité

La taille de l'intervalle est approximativement divisée par deux à chaque étape. Le nombre de comparaisons est donc

$$\mathcal{O}(\log n).$$

La recherche séquentielle, elle, peut nécessiter jusqu'à n comparaisons : sa complexité temporelle est $\mathcal{O}(n)$.

La dichotomie exige cependant que les données soient triées et que l'accès à un élément d'indice donné soit efficace.

5 Algorithme 3 : calcul d'une somme

5.1 Somme des premiers entiers

On veut calculer

$$S_n = \sum_{k=1}^{n} k.$$

```
def somme_entiers(n):
    s = 0
    for k in range(1, n + 1):
        s = s + k
    return s
```

Listing 3 – Calcul itératif d'une somme

5.2 Justification par invariant

Après l'itération d'indice k, l'accumulateur contient

$$s = \sum_{j=1}^{k} j.$$

En effet, avant la première itération, $s = 0$. Chaque itération ajoute le nouvel entier k à la somme précédente.

On obtient donc, à la fin,

$$S_n = \frac{n(n+1)}{2}.$$

5.3 Deux méthodes à comparer

Le calcul itératif effectue un nombre d'additions proportionnel à n : sa complexité est $\mathcal{O}(n)$.

La formule explicite utilise un nombre constant d'opérations arithmétiques, soit une complexité $\mathcal{O}(1)$ dans le modèle usuel à coût unitaire.

Cette comparaison illustre la différence entre une définition par récurrence et une expression explicite.

6 Algorithme 4 : méthode de dichotomie pour une équation

6.1 Principe

Soit f une fonction continue sur $[a, b]$, telle que $f(a)f(b) < 0$. Le théorème des valeurs intermédiaires garantit l'existence d'au moins une racine dans $[a, b]$.

On partage l'intervalle en deux, puis on conserve une moitié dont les valeurs aux extrémités sont de signes opposés.

```
def dichotomie(f, a, b, eps):
    fa = f(a)
    fb = f(b)

    if fa == 0:
        return a
    if fb == 0:
        return b
    if fa * fb > 0:
        raise ValueError("Changement de signe absent")
```

```
11
12      while b - a > eps:
13          m = (a + b) / 2
14          fm = f(m)
15
16          if fm == 0:
17              return m
18          elif fa * fm < 0:
19              b = m
20              fb = fm
21          else:
22              a = m
23              fa = fm
24
25      return (a + b) / 2
```

Listing 4 – Approximation d'une racine

6.2 Erreur et nombre d'itérations

Après k itérations, la longueur de l'intervalle est

$$\frac{b-a}{2^k}.$$

Le milieu de l'intervalle final est à une distance au plus

$$\frac{b-a}{2^{k+1}}$$

d'une racine contenue dans cet intervalle.

Pour garantir une erreur inférieure ou égale à ε, il suffit donc que

$$\frac{b-a}{2^{k+1}} \leq \varepsilon.$$

La méthode ne nécessite pas de calcul de dérivée. Elle est robuste, mais sa convergence est linéaire au sens où le nombre d'itérations nécessaire est logarithmique en $1/\varepsilon$.

7 Algorithme 5 : approximation de la racine carrée

7.1 Méthode de Héron

Pour $A > 0$, on cherche à approcher $\sqrt{A}$. La méthode de Héron repose sur la suite

$$u_{n+1} = \frac{1}{2}\left(u_n + \frac{A}{u_n}\right), \qquad u_0 > 0.$$

```
1  def racine_carree(A, eps=1e-12):
2      if A < 0:
3          raise ValueError("A doit etre positif")
4      if A == 0:
5          return 0.0
6
7      u = max(1.0, A)
8
9      while abs(u * u - A) > eps:
10         u = (u + A / u) / 2
11
12     return u
```

Listing 5 – Méthode de Héron

7.2 Propriétés

Pour $u > 0$, l'inégalité arithmético-géométrique donne

$$\frac{1}{2}\left(u + \frac{A}{u}\right) \geq \sqrt{A}.$$

Après une première étape, les termes sont donc supérieurs ou égaux à $\sqrt{A}$. Pour $u \geq \sqrt{A}$, on a également

$$\frac{1}{2}\left(u + \frac{A}{u}\right) \leq u.$$

La suite converge vers $\sqrt{A}$. À proximité de la limite, l'erreur est approximativement quadratique : la méthode de Newton est particulièrement rapide.

Remarque 7.1. *Le critère $|u^2 - A| \leq \varepsilon$ contrôle le résidu, pas directement l'erreur $|u - \sqrt{A}|$. Pour $u \geq 0$,*

$$|u - \sqrt{A}| = \frac{|u^2 - A|}{u + \sqrt{A}}.$$

En pratique, il convient de choisir un critère d'arrêt adapté à la précision recherchée et à l'échelle des données.

8 Algorithme 6 : tri par insertion

8.1 Principe

Le tri par insertion construit progressivement un préfixe trié. À chaque étape, il insère l'élément suivant à sa place dans la partie déjà ordonnée.

```
def tri_insertion(T):
    T = list(T)

    for i in range(1, len(T)):
        valeur = T[i]
        j = i - 1

        while j >= 0 and T[j] > valeur:
            T[j + 1] = T[j]
            j = j - 1

        T[j + 1] = valeur

    return T
```

Listing 6 – Tri par insertion

8.2 Correction

À la fin de l'itération d'indice i, le préfixe

$$T[0], T[1], \ldots, T[i]$$

est trié et contient les mêmes éléments que le préfixe initial de longueur $i + 1$.

L'initialisation est immédiate puisque le préfixe de longueur un est trié. L'étape d'insertion conserve l'ordre et la permutation des éléments. La propriété est donc vraie à chaque itération.

8.3 Complexité

Dans le pire cas, lorsque le tableau est initialement trié en ordre décroissant, le nombre de déplacements est proportionnel à

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}.$$

La complexité temporelle est donc $\mathcal{O}(n^2)$.

Sur des tableaux presque triés, le tri par insertion peut être très efficace. Sa complexité spatiale auxiliaire est $\mathcal{O}(1)$ si l'on exclut la copie initiale du tableau.

9 Algorithme 7 : calcul rapide d'une puissance

9.1 Exponentiation rapide

Pour calculer a^n, avec $n \in \mathbb{N}$, on peut utiliser

$$a^n = \begin{cases} 1 & \text{si } n = 0, \\ (a^{n/2})^2 & \text{si } n \text{ est pair}, \\ a(a^{(n-1)/2})^2 & \text{si } n \text{ est impair}. \end{cases}$$

```
def puissance(a, n):
    if n < 0:
        raise ValueError("n doit etre positif ou nul")

    resultat = 1

    while n > 0:
        if n % 2 == 1:
            resultat = resultat * a

        a = a * a
        n = n // 2

    return resultat
```

Listing 7 – Exponentiation rapide itérative

9.2 Correction et complexité

L'algorithme repose sur l'invariant

$$\texttt{resultat} \times a^n = a_{\text{initial}}^{n_{\text{initial}}},$$

où les valeurs de a et de n désignent celles de l'itération courante.

Si n est impair, un facteur a est transféré dans `resultat`. Le remplacement de a par a^2 et de n par $\lfloor n/2 \rfloor$ conserve l'invariant.

L'exposant est divisé par deux à chaque tour : le nombre de tours est $\mathcal{O}(\log n)$, au lieu de $\mathcal{O}(n)$ multiplications pour la méthode naïve.

Cette analyse compte les multiplications comme des opérations élémentaires. Pour des entiers de grande taille, le coût réel dépend aussi de la taille des nombres manipulés.

10 Comparaison des algorithmes

Le tableau suivant compare quelques méthodes en fonction de la taille n des données.

Algorithme	**Complexité temporelle**	**Condition ou intérêt**
Somme itérative	$\mathcal{O}(n)$	Méthode élémentaire, invariant simple
Recherche séquentielle	$\mathcal{O}(n)$	Fonctionne sans tri préalable
Recherche dichotomique	$\mathcal{O}(\log n)$	Nécessite des données triées et un accès efficace
Tri par insertion	$\mathcal{O}(n^2)$ au pire	Efficace pour de petites listes presque triées
Exponentiation rapide	$\mathcal{O}(\log n)$ multiplications	Réduit le nombre d'opérations
Algorithme d'Euclide	Nombre d'étapes logarithmique en la taille des entrées	Calcul du PGCD

Il faut distinguer le nombre d'opérations élémentaires de leur coût effectif. En particulier, les multiplications de grands entiers n'ont pas toutes le même coût.

11 Démarche de résolution d'un problème algorithmique

Une démarche rigoureuse peut être organisée en plusieurs étapes :

1. **Spécifier le problème :** préciser les entrées, les sorties et les hypothèses.
2. **Concevoir une méthode :** décomposer le problème et choisir les structures de contrôle adaptées.
3. **Écrire l'algorithme :** utiliser un langage naturel précis, un pseudo-code ou un langage de programmation.
4. **Tester :** considérer des cas ordinaires, limites et éventuellement des entrées invalides.
5. **Démontrer la correction :** établir un invariant, une propriété de récurrence ou une preuve adaptée.
6. **Démontrer la terminaison :** identifier une quantité entière positive qui décroît strictement, ou établir une borne sur le nombre d'itérations.
7. **Analyser la complexité :** compter les opérations en fonction de la taille des données.

Les tests ne remplacent pas une démonstration. Ils permettent cependant de détecter de nombreuses erreurs et d'éclairer la formulation des conjectures.

12 Exploitation pédagogique

L'algorithmique est un domaine privilégié pour articuler mathématiques, programmation et raisonnement.

Au collège et au lycée, on peut faire travailler les élèves sur :

— les variables et les affectations ;

— les instructions conditionnelles et les boucles ;

— la simulation d'une suite définie par récurrence ;

— la recherche d'un seuil ou d'une approximation ;

— la comparaison expérimentale de deux méthodes ;

— la justification de la correction d'un programme.

La progression peut commencer par la simulation manuelle, se poursuivre par l'écriture d'un pseudo-code, puis aboutir à l'implémentation et à la validation.

Un même problème peut être résolu par plusieurs algorithmes. Leur comparaison aide à comprendre qu'un programme ne se réduit pas à une suite d'instructions qui fonctionne sur quelques exemples : il doit répondre à une spécification et être justifié.

13 Liste d'exercices

Les exercices suivants peuvent servir à l'entraînement, à une séance de travaux pratiques ou à un prolongement oral. Les niveaux sont indicatifs.

Exercice 1 – Maximum d'une liste

Écrire un algorithme qui renvoie le maximum d'une liste non vide d'entiers.

a) Justifier l'initialisation de la variable maximum.

b) Proposer un invariant de boucle.

c) Déterminer le nombre de comparaisons effectuées.

d) Que faut-il modifier pour renvoyer aussi un indice d'occurrence du maximum ?

Exercice 2 – Recherche d'un élément

Écrire une recherche séquentielle dans une liste. Comparer sa complexité à celle de la recherche dichotomique. Préciser pourquoi le tri préalable peut modifier le coût total d'une recherche répétée.

Exercice 3 – PGCD et Bézout

Appliquer l'algorithme d'Euclide aux entiers 414 et 252.

a) Calculer leur PGCD.

b) Remonter les divisions euclidiennes pour obtenir une identité de Bézout.

c) Écrire un programme calculant également les coefficients de Bézout.

Exercice 4 – Somme d'une suite géométrique

Écrire un algorithme calculant

$$S_n = \sum_{k=0}^{n} q^k$$

pour un réel q et un entier $n \geq 0$. Comparer l'algorithme itératif à la formule

$$S_n = \frac{1 - q^{n+1}}{1 - q}, \qquad q \neq 1.$$

Traiter séparément le cas $q = 1$.

Exercice 5 – Approximation de $\sqrt{2}$

Mettre en œuvre la méthode de Héron avec $u_0 = 1$.

a) Calculer les premiers termes.

b) Étudier la convergence de la suite.

c) Comparer l'erreur absolue et le résidu.

d) Choisir un critère d'arrêt garantissant une précision donnée.

Exercice 6 – Dichotomie

Approcher une solution de l'équation

$$x^3 + x - 1 = 0$$

sur l'intervalle $[0, 1]$ à 10^{-4} près. Justifier l'existence d'une solution et déterminer une borne sur le nombre d'itérations nécessaires.

Exercice 7 – Tri

Implémenter le tri par insertion.

a) Démontrer sa correction.

b) Compter les comparaisons dans le meilleur et le pire cas.

c) Comparer ses résultats à ceux d'un tri intégré.

d) Étudier l'effet d'un tableau déjà trié.

Exercice 8 – Exponentiation rapide

Écrire une version récursive de l'exponentiation rapide. Démontrer sa correction par récurrence sur l'exposant. Comparer le nombre de multiplications à celui de la méthode naïve pour plusieurs valeurs de n.

Exercice 9 – Suite définie par récurrence

On considère

$$u_0 = 1, \qquad u_{n+1} = \frac{1}{2}\left(u_n + \frac{2}{u_n}\right).$$

Écrire un programme calculant u_n, puis un programme s'arrêtant lorsque deux termes successifs diffèrent de moins de ε. Discuter les garanties apportées par ce critère d'arrêt.

Exercice 10 – Comparaison de méthodes

Pour un entier n, comparer le calcul de $n!$ par une boucle et par récursion.

a) Démontrer la correction des deux programmes.

b) Comparer leurs complexités temporelles.

c) Comparer leur utilisation de la mémoire.

d) Discuter les risques liés aux limites numériques et à la profondeur de récursion.

14 Questions possibles du jury

— Quelle différence faites-vous entre un algorithme et un programme ?

— Comment démontrez-vous qu'une boucle termine ?

— Qu'est-ce qu'un invariant de boucle ?

— Pourquoi un algorithme correct peut-il être peu efficace ?

— Dans quelles conditions la recherche dichotomique est-elle pertinente ?

— Comment distinguer une preuve mathématique d'une validation expérimentale ?

— Comment enseigner l'algorithmique sans réduire l'activité mathématique à la programmation ?

— Comment les erreurs d'arrondi peuvent-elles modifier le comportement d'un algorithme numérique ?

15 Conclusion

Les exemples présentés montrent la diversité des méthodes algorithmiques : calcul exact en arithmétique, recherche, tri, approximation numérique et calcul récursif.

L'étude d'un algorithme ne se limite pas à son exécution. Elle comprend la formulation précise du problème, la preuve de correction, l'établissement de la terminaison et l'analyse de la complexité.

Ces questions offrent un lien naturel entre raisonnement mathématique, expérimentation informatique et enseignement. La comparaison de plusieurs algorithmes pour un même problème est particulièrement féconde pour faire apparaître le rôle des hypothèses, des invariants et des choix de conception.

Références

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest et C. Stein, *Introduction to Algorithms*, 4e édition, MIT Press, 2022.

[2] D. E. Knuth, *The Art of Computer Programming*, vol. 1 : *Fundamental Algorithms*, 3e édition, Addison-Wesley, 1997.

[3] P. Robert, *Mathématiques discrètes*, référence générale pour les raisonnements discrets et l'analyse combinatoire.

[4] Python Software Foundation, *Python 3 Documentation*, documentation officielle du langage Python. https://docs.python.org/3/

[5] Ministère de l'Éducation nationale, *Programmes d'enseignement de mathématiques du collège et du lycée*, ressources institutionnelles disponibles sur Éduscol. https://eduscol.education.fr/

[6] Euclide, *Les Éléments*, livre VII, pour les fondements de l'arithmétique et les propriétés liées à la divisibilité.