EA4 – Éléments d'algorithmique
TD n° 1 : opérations arithmétiques

Exercice 1 : des limites du progrès...

Il y a 25 ans, un ordinateur faisait dix millions d'opérations par seconde et implémentait un algorithme de test de primalité demandant $1200\sqrt{n} + 100$ opérations pour un tableau d'entrée de taille n. On souhaite le comparer à un ordinateur 100 fois plus rapide, mais sur lequel tourne un (moins bon) algorithme de test de primalité demandant $\frac{n}{2}$ opérations. Quels sont les temps de calcul de chacun pour une entrée de taille $n = 10^8$? et $n = 10^{12}$? Quelle devrait être l'accélération pour avoir le même temps de calcul dans chacune des deux configurations ?

Exercice 2 : exponentiation binaire, ou *exponentiation rapide*

1. Décrire un algorithme récursif `exponentiation_binaire_rec(m, n)` permettant de calculer m^n, similaire à l'algorithme vu en cours, mais basé sur l'identité $m^{2k} = (m^2)^k$ plutôt que $m^{2k} = (m^k)^2$.
2. Dérouler cet algorithme pour `m = 2, n = 13`.
3. Combien d'appels récursifs sont effectués en fonction de `n` ? Et combien de multiplications dans le pire des cas ?
4. Proposer une version itérative de cet algorithme.

Exercice 3 : évaluation de polynômes

1. Proposer un algorithme le plus naïf possible qui, étant donné un polynôme P et une valeur x, calcule $P(x)$. On supposera que P est décrit par un tableau contenant, en case d'indice i, le coefficient du monôme de degré i.
2. Appliquer votre algorithme au polynôme $P(x) = x^4 + x^3 + x^2 + x + 1$ avec $x = 3$. Que constatez-vous ?
3. Quel est le nombre d'additions et de multiplications effectuées lors de l'exécution de cet algorithme sur un polynôme de degré n ?
4. Comment peut-on améliorer la complexité de cet algorithme en utilisant l'exponentiation binaire ?
5. Comment peut-on améliorer la complexité de cet algorithme en conservant dans une variable entière les puissances successives de la valeur de x ?
6. Montrer que l'algorithme suivant résout le même problème :

```
def horner(P, x) :
  res = 0
  for coeff in P[::-1] :      # effectue un parcours du tableau à l'envers
    res = res * x + coeff
  return res
```

Quelle est le nombre d'additions et de multiplications d'entiers effectuées lors de l'exécution de cet algorithme sur un polynôme de degré n ?

Exercice 4 : racine carrée

Pour cet exercice, nous considérons les deux algorithmes suivants :

```
def racine_1(n) :
  i = 0
  carre = i * i
  while carre <= n :
    if carre == n :
      return i
    i = i + 1
    carre = i * i
  return i - 1
```

```
def racine_2(n) :
  i, j = 0, n
  if n < 2 : return n
  while j > i + 1 :
    m = (i + j) // 2
    carre = m * m
    if carre == n : return m
    elif carre < n : i = m
    else : j = m
  return i
```

1. Tester `racine_1` pour les valeurs `n = 9` et `n = 14`.
2. Montrer que `racine_1` calcule $\lfloor \sqrt{n} \rfloor$.
3. Calculer le nombre d'additions et de multiplications au pire lors de l'exécution de cet algorithme en fonction de `n`.
4. Montrer que `racine_2` calcule aussi $\sqrt{n}$, et calculer le nombre d'additions et de multiplications au pire en fonction de n.
5. Pour quelles valeurs de n est-il préférable d'utiliser chacun de ces algorithmes ?

Exercice 5 : nombres premiers

1. Proposer un algorithme `est_premier(p)` *le plus naïf possible* permettant de déterminer si un entier `p` est premier. Quelle est sa complexité ? Comment peut-on l'améliorer ?

On considère maintenant l'algorithme suivant :

```
def eratosthene(n) :
  tab = [False, False] + [True]*(n-1) # tab = [False,False,True,True,...,True]
  for i in range(2, n+1) :
    if tab[i] :
      for k in range(2*i, n+1, i) : # depuis 2i, par pas de i, sans dépasser n
        tab[k] = False
  return tab
```

2. Exécuter l'algorithme pour `n = 10`.
3. Que représente le tableau calculé par `eratosthene(n)` ? Justifier.
4. Calculer un majorant du nombre d'additions et de multiplications d'entiers effectuées par l'algorithme pour une entrée `n`.
5. Pour vous convaincre de l'impact pratique des différences de complexité, comparer les temps de calcul de `[ p for p in range(10**6) if est_premier(p) ]` et de `[ p for p,b in enumerate(eratosthene(10**6)) if b ]`.