Exercices INF7440 : série #10

0. Algorithmes d'exploration des graphes

a. Soit un graphe non orienté G. Peut-on utiliser, directement ou en le modifiant, un des algorithmes d'exploration de graphes pour déterminer les composantes connexes de G?

   Rappel : les composantes *connexes* d'un graphe sont les groupes de sommets qui sont reliés les uns autres, directement ou indirectement.

b. On veut générer un ensemble de valeurs à l'aide de certaines opérations arithmétiques. On part tout d'abord de la valeur 1. Pour construire d'autres valeurs, on peut, à partir d'une valeur existante : (i) multiplier la valeur par 2 ; (ii) si la valeur est supérieure à 2, effectuer une division entière par 3 (en ignorant la partie fractionnaire).

   Le graphe de la Figure 1 représente les différentes valeurs qui peuvent être obtenues, à partir de la valeur de départ 1, en effectuant uniquement les opérations permises — les arcs vers l'arrière ne sont pas indiqués.

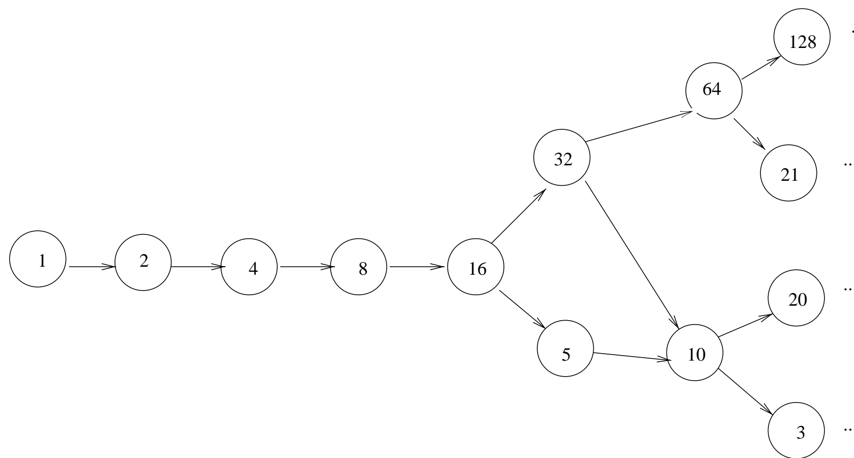

Figure 1: Graphe de multiplications par 2 et divisions par 3

   Expliquez comment faire pour que, si un nombre donné k peut être produit à l'aide d'une telle suite d'opérations, on soit certain de pouvoir le déterminer (i.e., répondre de façon affirmative) — mais s'il n'existe pas de telle suite, alors l'algorithme peut ne rien répondre.

1. Algorithme pour le problème du sac-à-dos 0–1

L'algorithme 1 est un algorithme récursif d'exploration exhaustive des solutions pour le problème du sac-à-dos 0–1. Rappelons que dans cette version du problème (donc contrairement à la version fractionnaire), un item doit être pris au complet ou ne pas être pris du tout.

La procédure principale de l'algorithme est `BeneficeSac`, qui elle appelle ensuite la procédure `BeneficeSacRec`. Pour simplifier le problème, comme au chapitre 2 ("Diviser-pour-régner"), on cherche uniquement à calculer le bénéfice maximum résultant. Toujours pour simplifier le problème, on suppose aussi (voir l'assertion au début de la procédure `BeneficeSac`) que les items sont *triés* en ordre décroissant de rapport bénéfice par unité de poids.

a. Quel est le temps d'exécution (complexité asymptotique) de cet algorithme? Pour simplifier l'analyse, vous pouvez *ignorer* les coûts associés au calcul du `benefice` d'une solution donnée et vous pouvez supposer que les opérations sur les ensembles s'effectuent toutes en temps $\Theta(1)$ — on s'intéresse donc uniquement au nombre de noeuds de l'arbre des appels.

b. Supposons que l'on fasse les modifications suivantes :

- On remplace la condition de fin de récursion "`cardinalite(vus) == n`" par la condition "`maximum(vus) == n`".
- On remplace la boucle `for` dans la procédure `BeneficeSacRec` par la boucle suivante, c'est-à-dire qu'on remplace donc la condition "`~estElement(vus, i)`" par la condition "`i > maximum(vus)`" :

```
for [i = 1 to n st i > maximum(vus)] {
  ...
}
```

(i) Que devient alors le temps d'exécution (complexité asymptotique) de cet algorithme (en supposant les mêmes simplifications que pour l'exercice précédent)?

(ii) Expliquez pourquoi le résultat produit est quand même correct.

c. De quelle façon pourrait-on améliorer encore cet algorithme? Expliquez brièvement (grandes lignes) les modifications qu'il faudrait apporter au code de ces procédures pour les améliorer.

Remarque importante : Examinez attentivement la précondition de la procédure `BeneficeSac`.

Remarque : Les ensembles utilisés dans l'algorithme 1, de type `Ensemble`, sont du même type que ceux utilisés pour la solution de programmation dynamique au problème du commis voyageur. Toutefois, on suppose qu'une opération supplémentaire a été définie sur les ensembles :

```
procedure maximum( Ensemble s ) returns int leMax
# POSTCONDITION
#    estVide(s) => leMax = 0
#   ~estVide(s) => leMax = MAXIMUM( i SUCH THAT i IN s :: i )
```

```
procedure benefice( Ensemble e, int b[*] ) returns int benef
# POSTCONDITION
#   benef = somme des benefices associes aux items dans l'ensemble e.
{
  benef = 0;
  for [i = 1 to ub(b)] {
    if ( estElement(e, i) ) { benef += b[i] }
  }
}

procedure BeneficeSacRec( int W, int p[*], int b[*],
                          Ensemble vus, Ensemble choisis, int poidsRestant,
                          ref int benefMax
                         )
{
  int n = ub(p);

  if (cardinalite(vus) == n) {
    # Tous les elements ont ete examines. On met a jour le benefice maximum.
    benefMax = max( benefMax, benefice(choisis, b) );
  } else {
    # Il reste des elements a examiner. ...
    for [i = 1 to n st ~estElement(vus, i)] {
      if ( p[i] <= poidsRestant ) {
        # Il reste de la place pour inclure l'item.
        BeneficeSacRec( W, p, b,
                        ajouter(vus, i), ajouter(choisis, i), poidsRestant-p[i],
                        benefMax );
      } else {
        # Il ne reste pas de place pour inclure l'item.
        BeneficeSacRec( W, p, b,
                        ajouter(vus, i), choisis, poidsRestant,
                        benefMax );
      }
    }
  }
}


procedure BeneficeSac( int W, int poids[*], int benefs[*] ) returns int benef
# PRECONDITION
#   Les tableaux poids et benefs sont de meme taille
#   Les tableaux sont ordonnes selon leur rapport benefice/poids
{
  int n = ub(poids);
  benef = 0;
  BeneficeSacRec( W, poids, benefs, creerVide(n), creerVide(n), W, benef )
}
```

Algorithme 1: Algorithme avec marche arrière pour le problème du sac-à-dos 0–1