Algorithmes pour les graphes - TD sur plateforme
Exercice 1 : Énumérer toutes les permutations d'un ensemble

Étant donné un graphe orienté $G = (S, A)$ et une fonction associant un coût à chaque arc de G, le problème du voyageur de commerce consiste à chercher le plus court circuit passant par chaque sommet de S, la longueur d'un circuit étant définie par la somme des coûts de ses arcs. Ce problème est $\mathcal{NP}$-difficile, et nous allons programmer différents algorithmes pour le résoudre. Nous allons supposer pour cela que le graphe est complet. En effet, si G n'est pas complet, nous pouvons facilement le transformer en un graphe complet admettant la même solution pour le voyageur de commerce : il suffit de fixer le coût de chaque arc ajouté à une valeur très grande (par exemple nk, où n est le nombre de sommets et k est le coût de l'arc le plus long de G).

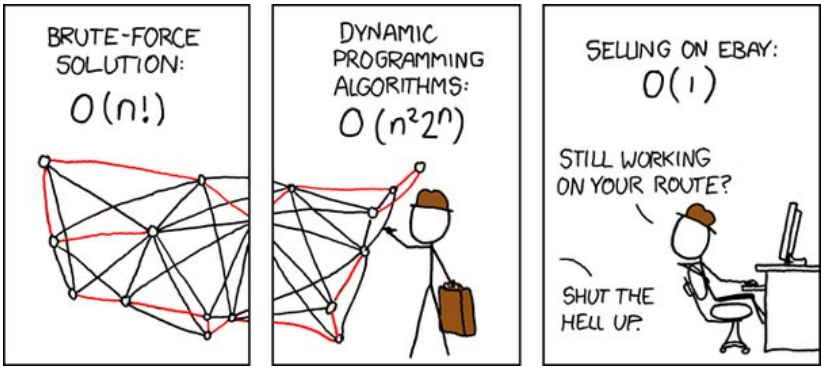

Image empruntée à xkcd.org

Le premier exercice consiste à énumérer toutes les permutations de sommets possibles, comme décrit dans la procédure récursive suivante :

Procédure *permut*$(vus, nonVus)$
  Entrée : Une liste ordonnée de sommets vus (déjà visités)
    Un ensemble de sommets $nonVus$ (restant à visiter)
  Postcondition : Affiche toutes les listes commençant par vus et se terminant par une permutation de $nonVus$
  si $nonVus$ *est vide* **alors** Afficher les éléments de vus, dans l'ordre de la liste;
  pour *chaque sommet* $s_j \in nonVus$ **faire**
    Ajouter s_j à la fin de la liste vus et enlever s_j de l'ensemble $nonVus$
    permut$(vus, nonVus)$
    Retirer s_j de la fin de la liste vus et remettre s_j dans l'ensemble $nonVus$

Pour éviter d'énumérer plusieurs fois chaque circuit (une fois pour chaque sommet de départ), nous pouvons n'énumérer que les permutations commençant par un sommet donné (par exemple, le sommet 0). Dans ce cas, au premier appel, la liste vus doit contenir un seul sommet (0), tandis que l'ensemble $nonVus$ doit contenir tous les autres sommets de S.

Votre travail : Vous devez implémenter la procédure `permut` (implémentant l'algorithme *permut* décrit ci-dessus) :

```
void permut(int vus[], int nbVus, int nonVus[], int nbNonVus)
```

telle que :
- le tableau `vus[0..nbVus-1]` contient les sommets déjà visités (ceux de la liste vus) ;
- le tableau `nonVus[0..nbNonVus-1]` contient les sommets qui n'ont pas encore été visités (ceux de l'ensemble $nonVus$).

Cette procédure affiche toutes les séquences de sommets commençant par `vus[0..nbVus-1]`, et se terminant par n'importe quelle permutation de `nonVus[0..nbNonVus-1]`.

Code fourni (téléchargeable sur `http://liris.cnrs.fr/csolnon/TPTSP/code1.c`**) :** Procédure `main` appelant votre procédure `permut`, et procédure `printSolution` affichant les sommets de `vus[0..nbVus-1]` (cette procédure vous est fournie pour éviter les problèmes de formatage des sorties).

```
#include <stdio.h>
#include <stdlib.h>

void printSolution(int vus[], int nbVus){
    /* Postcondition : Affiche les valeurs de vus[0..nbVus-1] separees par un espace,
       suivies par un retour à la ligne */
    int i;
    for (i=0; i<nbVus; i++)
        printf("%d ",vus[i]);
    printf("\n");
}

void permut(int vus[], int nbVus, int nonVus[], int nbNonVus){
    /*
    Entree :
     - vus[0..nbVus-1] = sommets visites
     - nonVus[0..nbNonVus-1] = sommets non visites
    Precondition : nbVus > 0 et vus[0] = 0 (le tour commence toujours par le sommet 0)
    Postcondition : affiche tous les tours commençant par vus[0..nbVus-1] et se terminant
                    par les sommets de nonVus[0..nbNonVus-1] (dans tous les ordres possibles)
     */
    // INSEREZ VOTRE CODE ICI !
}

int main(){
    int nbSommets, i;
    scanf("%d",&nbSommets);
    int vus[nbSommets], nonVus[nbSommets-1];
    vus[0] = 0;
    for (i=0; i<nbSommets-1; i++)
        nonVus[i] = i+1;
    permut(vus,1,nonVus,nbSommets-1);
    return 0;
}
```

Exemple d'exécution : Si l'entier $nbSommets$ saisi en entrée est 4, alors le programme affichera les 6 permutations suivantes (l'ordre dans lequel les permutations sont affichées peut varier) :

```
0 1 3 2
0 1 2 3
0 2 1 3
0 2 3 1
0 3 1 2
0 3 2 1
```