CIR2 - Algorithmique Avancée

Sujet TD&P n°4 - Séances n° 5 & 6

Du scénario à l'analyse de résultat,
étude d'un

algorithme combinatoire

pour le problème du

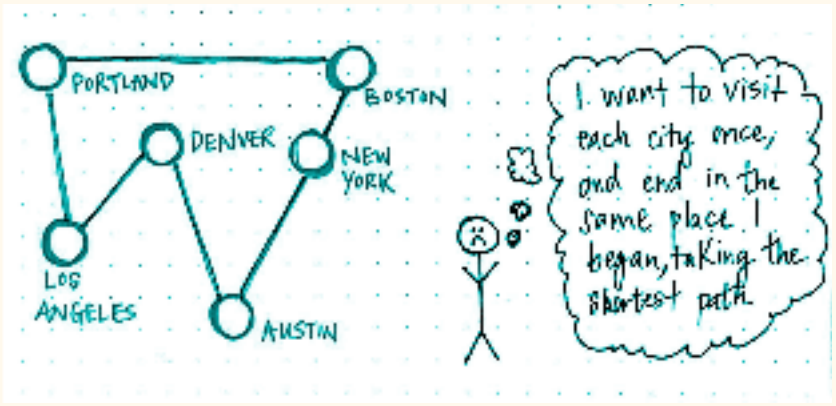
voyageur de commerce

PARTIE 1
Introduction

Résumé & Objectifs.

- **Les objectifs.** Comprendre et implémenter un algorithme récursif permettant de résoudre le problème d'optimisation **du voyageur de commerce**.
- **Le problème.** Trouver un cycle hamiltonien (chemin cyclique qui passe par chaque ville une seule fois) dont la longueur totale est la plus petite possible.
- **L'approche.** Nous allons mettre en place un algorithme de type 'brute force' qui consiste à lister toutes les solutions, les évaluer et les comparer pour en retirer la meilleure. Il s'agit de faire le lien entre le problème d'optimisation, son algorithme de résolution, le scénario considéré (i.e., les données d'entrée du problème) et le traitement des sorties.

Ce TD&P fait le pont entre les rappels algorithmiques/C++ et la résolution de problèmes difficiles. Il comporte beaucoup de questions de code mais requiert une attention toute particulière sur la compréhension d'un algorithme. Ceux qui auront réussi à lier ce problème à l'algorithme étudié lors de la toute première séance, verront que la principale différence ici, est la prise en compte des distances entre chaque ville. Ces dernières permettent l'évaluation de la qualité de chaque solution.



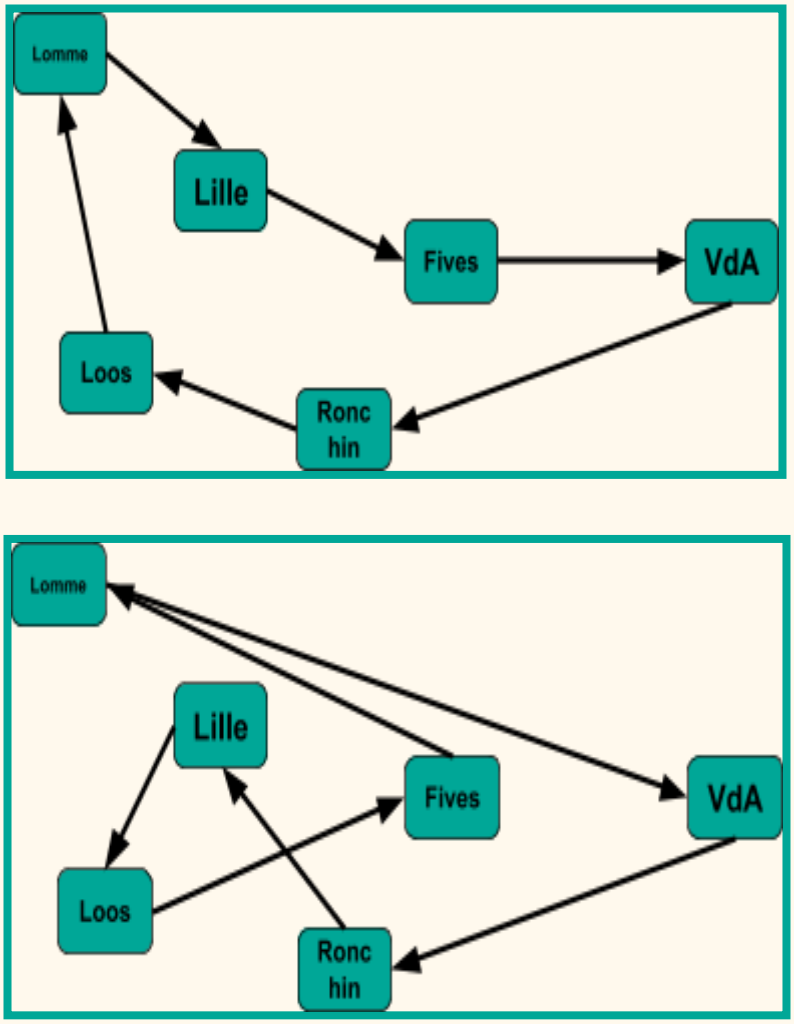
Pic : https://medium.com/basecs/the-trials-and-tribulations-of-the-traveling-salesman-56048d6709d

Samuel Deleplanque. (prénom.nom@junia.com)

Enoncé du problème. On peut définir le problème du voyageur de commerce comme suit :

- A partir d'une liste de villes et une matrice des distances les reliant, on cherche à déterminer le ou les cycles[1] les plus courts passant une seule fois par chaque ville.

Exemple. Les graphiques ci-dessous représentent un cas avec 6 villes et 2 cycles potentiels.

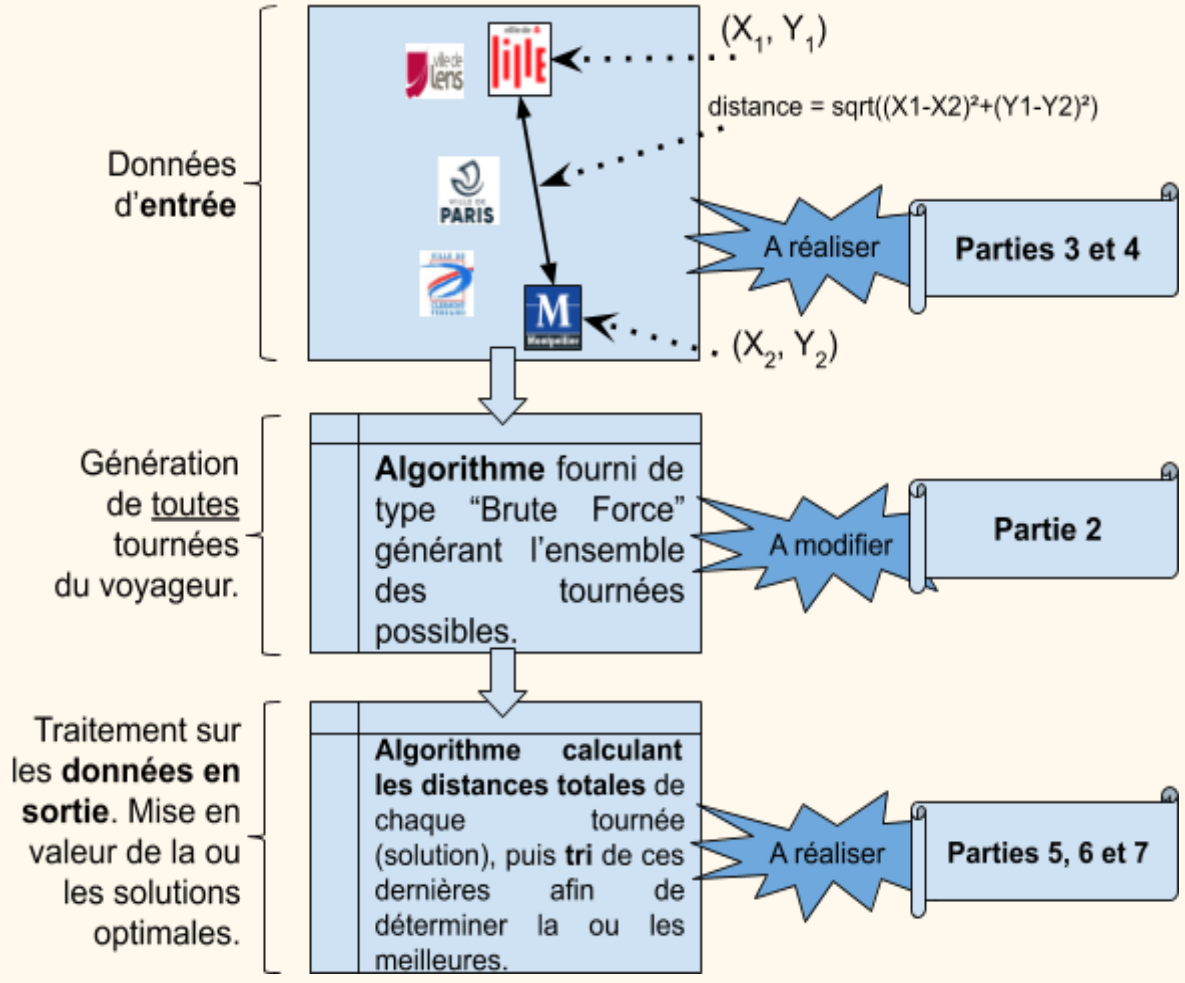


Représentation de 2 des 120 cycles pour un scénario à 6 villes.

Question. Combien de tournées existe-t-il pour un scénario avec 6 villes ?

[1] Chaque solution ne propose qu'un seul cycle. En effet, sur l'exemple donné, on aurait pu imaginer un premier cycle (Loos-Lomme-Lille) et un second (Fives-VdA-Ronchin), mais cela ne satisferait pas toutes les contraintes du problème. L'expression "Les cycles" veut ici exprimer le fait qu'il peut exister plusieurs solutions optimales, c'est-à-dire plusieurs solutions avec la même plus courte distance totale.

Déroulement du TD&P. Après quelques adaptations de l'algorithme principal, nous allons développer une méthode qui produira des scénarios (i.e., noms des villes et de leurs coordonnées). L'algorithme principal (fourni) va ainsi générer l'ensemble des tournées possibles du voyageur de commerce à partir de ces scénarios. Ensuite, nous allons développer une autre méthode pour le traitement des sorties. Il sera alors possible de trier les tournées selon la distance totale, permettant ainsi de déterminer la ou les tournées optimales (i.e., de plus petite distance totale). La figure ci-dessous tente de résumer les différentes parties de TD&P.



Points principaux du TD&P

PARTIE 2
Manipulation d'un code existant.

- **Algo.** Faites la trace (à la main) de l'**algorithme récursif** implémenté dans `toutesLesPermutations` pour l'argument `"AB"` fourni dans le `main()` du programme c++ suivant :

```
//============================================================================
/// \name   allPermutationWithSTL.cpp
/// \version    1.1 (2020.2021)
/// \brief    Algorithme combinatoire récursif
//============================================================================
/* Remplacez l'include suivant par tous les #include de la STL nécessaires si
vous utilisez Visual Studio */
#include <bits/stdc++.h>
using namespace std;
void toutesLesPermutations(string & villes, int debut, int fin);
int main()
{
  string villes = "AB";
  toutesLesPermutations(villes, 0, villes.size() - 1);
  return 0;
}
/// \brief Methode recursive generant l ensemble des mots possibles
/// avec les caractères du mots villes donne en parametre.
void toutesLesPermutations(string & villes, int debut, int fin)
{
  if (debut == fin)
      cout << villes << endl;
  else
  {
      // Permutations made
      for (int i = debut; i <= fin; i++)
      {
          // echange des deux lettres
          swap(villes[debut], villes[i]);
          // Appel Recursif
          toutesLesPermutations(villes, debut+1, fin);
          // On revient à la situation précédente
          swap(villes[debut], villes[i]);
      }
  }
}
```

- **C++. Compiler le code puis tester des entrées différentes.** Après compilation, testez différentes entrées comme par exemple :

```
string villes = "ABCD";
string villes = "ABDEF";
string villes = "ALGORITHME";
```

Ce dernier cas vous rappellera le TD&P1 mais avec une nouvelle manière de générer toutes les combinaisons. Ne soyez pas trop patient...

- **C++. Modifiez le programme[2] dans le but de manipuler un vecteur de noms de villes.** Dans le code ci-dessus, `'string villes = "AB"'` consistait à n'utiliser qu'une seule chaîne de caractères, et les combinaisons étaient faites sur les lettres. Modifiez votre code afin d'utiliser plusieurs chaînes de caractères et faire les combinaisons sur celles-ci. Vous pouvez utiliser le vecteur suivant :

  - `vector<string> vecteurDeNomsDeVille;`

  Ainsi, si vous déclarez `vector<string> vecteurDeNomsDeVille{ "Lille", "Vda"}`, le programme devrait afficher `"Lille Vda"` puis `"Vda Lille"`.

[2] Ce travail d'adaptation d'un algorithme est une activité récurrente dans la vie d'un développeur.

PARTIE 3
Génération des noms de villes.

Objectif. Nous allons générer un scénario du problème du voyageur de commerce grâce à la méthode `rand`[3]. Une ville sera définie par son nom et sa localisation sur 1 plan en 2 dimensions (avec ses coordonnées X et Y).

- **C++. Intégrez les constantes entières et nécessaires à ce travail[4] :**

```
constexpr int codeASCIIde_a = 97;
constexpr int codeASCIIde_A = 65;
constexpr int nombreDeLettres = 26;
constexpr int tailleMinNomVille = 4;
constexpr int tailleMaxNomVille = 12;
constexpr int grainePourLeRand = 1;
constexpr int nombreDeVilles = 4;
constexpr int nombreCombinaisons = 24;
constexpr int tailleCoteCarte = 100;
```

- **C++. Générez les noms des villes de** `vecteurDeNomsDeVille`. Utilisez la méthode `rand` pour chaque lettre de chaque nom de ville afin de générer à chaque fois un entier qui sera transformé en lettre grâce au code ASCII.
  Seule la première lettre doit être en majuscule. A vous de générer en amont la taille du nom qui devra se trouver entre 4 et 10 caractères[5]. On pourra par exemple générer le nombre de lettres de chaque ville avec :
  - `nbLettresNomVille = tailleMinNomVille + rand()%(tailleMaxNomVille-tailleMinNomVille+1);`

Vous devriez pouvoir facilement répondre à cette question avec deux boucles `for` imbriquées : une pour chaque ville, et une autre pour chaque lettre minuscule de chacune de ces villes. Si vous bloquez sur cette étape, vous êtes autorisés à écrire les villes “en dur”, i.e., directement dans le code, quitte à revenir dessus par la suite.

Note. Ces mêmes noms serviront plus tard de clés dans une std::map[6] afin de récupérer des informations sur chaque ville.

[3] Vous pouvez -ici et plus tard- bien entendu utiliser d’autres méthodes de génération de nombres pseudo-aléatoires provenant de la STL que vous avez pu voir en cours de C++. Pour rand, vous pouvez vous aider de http://www.cplusplus.com/reference/cstdlib/rand.

[4] Vous pouvez passer outre ces constantes en utilisant les paramètres d’exécution de votre programme. Utilisez la méthode qui vous fait perdre le moins de temps.

[5] Vous pouvez aller plus loin : générez l’existence (ou pas) de ‘-’, de nom de types “Boulogne-sur mer”, “Saint-Flour”, etc. **Il y aussi la possibilité de tirer aléatoirement parmi un ensemble de syllabes afin de rendre le nom des villes beaucoup plus agréable à lire.**

[6] Rappel : une `std::map` est constituée de multiples `std::pair<clé, “valeurs”>`. Si la clé est ici un `std::string` (`std::pair<std::string, “valeurs”>`), on utilisera le nom de la ville pour avoir accès aux valeurs associées que l’on définit plus tard dans le sujet.

PARTIE 4

Génération de la localisation des villes et utilisation des <map> et <tuples>

- **C++. Générez la localisation de chaque ville.** Pour chacune des n villes, et donc toujours dans la boucle `for` qui génère leur nom, nous allons générer une localisation sur un plan en 2 dimensions. Chaque ville aura alors une coordonnée X et une coordonnée Y. Nous allons pour cela imaginer une carte carrée dont la taille du côté est définie par la constante `tailleCoteCarte`.
  De nouveau grâce à la méthode `rand`(), générez ainsi les X et Y (en entier) pour chacune de ces villes. Une fois les coordonnées générées, celles-ci seront accompagnées de l'indice[7] de la ville (vous pouvez utiliser celui de la boucle `for` associée) dans un nouveau conteneur du C++11 : les tuples[8] :
  - https://en.cppreference.com/w/cpp/utility/tuple

  On génère pour chaque ville un tuple qui contiendra `<l'index de la ville, la coordonnée X, la coordonnées Y>` :

```
- auto monTuple = std::make_tuple(i, X, Y);
```

  Vous trouverez dans la documentation des tuples comment avoir accès aux différents éléments de ces derniers ou sur :
  - https://www.geeksforgeeks.org/tuples-in-c/

- **C++. Rassemblez les tuples et les noms de chaque ville dans une `std::map`** où la 'clé' est le nom de la ville et la 'valeur' est le tuple associé. On pourra déclarer la `std::map` de la manière suivante :

```
- std::map<std::string,std::tuple<int,int,int> >
  maMapNomsVillesEtCoordonnees;
```

  et ajoutez chaque nouvelle ville -et donc nouveau tuple- de cette manière :

```
maMapNomsVillesEtCoordonnees.insert(
      maMapNomsVillesEtCoordonnees.begin(),
      std::pair<string, tuple<int,int,int> >(monStringTemp,
                                  tempTuple_indexVille_X_Y));
```

  Avec `monStringTemp` le nom de la ville et `tempTuple_indexVille_X_Y` le tuple qui lui est associé.

[7] Cet indice doit être unique et débuter à 0. Avec `n` villes, on aura `i = 0..n-1`. Ces indices seront utilisés pour identifier les villes dans la matrice des distances.

[8] Dans cet exemple, vous allez utiliser un `tuple` avec 3 entiers, ce qui justifie à peine l'utilisation d'une telle structure. Il faut garder en tête que les types utilisés dans les `tuples` peuvent tous être différents, et le nombre d'éléments également. Ils peuvent être très pratiques sur des projets plus importants.

- **C++. Développez une méthode qui va afficher tout le contenu de la `map`.** Si `it` est un itérateur sur votre map, l'accès à la clé peut se faire avec `it->first` et l'accès à la valeur (ici : le tuple) est réalisé avec `it->second`. Ainsi, vous allez devoir afficher le contenu du tuple `it->second`, ce qui est quelque peu déroutant. Par exemple, l'affichage de la coordonnée Y se fait de la manière suivante :

  - `std::cout << get<2> (it->second)`

  puisque Y est en 3ème position du tuple.

Remarque. Si vous pouvez configurer votre compilateur pour développer du `C++17` (ou alors s'il est déjà configuré comme tel), vous pouvez utiliser une méthode d'accès aux données d'un tuple plus intuitive comme dans l'exemple -pris au hasard- ci-dessous[9] :

```
// les variables gpa2, grade2, et name2 récupèrent directement
// les 3 valeurs du tuple en exemple et auto affecte le
// le bon type.
auto [ gpa2, grade2, name2 ] = get_student(2);
std::cout << "ID: 2, "
          << "GPA: " << gpa2 << ", "
          << "grade: " << grade2 << ", "
          << "name: " << name2 << '\n';
```

- **Question.** Quel impact a l'utilisation d'une `std::map` et non d'une `std::multimap` ?

[9] Code tiré de https://en.cppreference.com/w/cpp/utility/tuple.

PARTIE 5

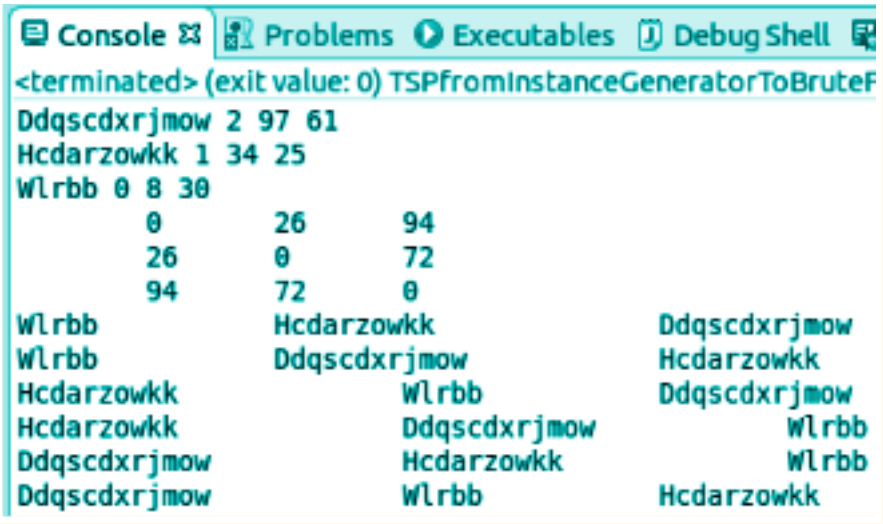
Génération des distances et utilisation de <cmath>.

- **C++. Implémentez une matrice des distances.** Les villes étant toutes générées, nous allons maintenant calculer toutes les distances qui les séparent et les sauvegarder dans un tableau 2D d'entiers que l'on nommera `DIST`. Vous avez le choix entre un double `<vector>` (pour lequel essayez de réserver la mémoire en amont, par exemple avec `reserve`) et un double `<array>`.
- **C++. Calculez les distances grâce à une méthode de `<cmath>`**. Ainsi, la distance entre deux villes de coordonnées (X_1, Y_1) et (X_2, Y_2) peut être calculée avec la norme suivante :
  - $\sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}$
  - Recherchez la bonne méthode dans http://www.cplusplus.com/reference/cmath/
  - Minimiser le nombre d'instructions élémentaires, en évitant, si possible, les calculs redondants tels que ceux liés à la symétrie.

PARTIE 6

Affichage des données générées.

- **C++. Affichez l'index, les coordonnées de chaque ville, la matrice des distances** `DIST`, **puis toutes les permutations des villes** (en choisissant un `n` suffisamment petit de façon à ce qu'il permette l'affichage des permutations en quelques petites secondes). Ainsi, si "Wlrbb", "Ddqscdxrjmow" et "Hcdarzowkk" sont de magnifiques noms de villes pour `n = 3`, on devrait obtenir l'affichage suivant :



PARTIE 7
Évaluation d'une solution.

- **C++. Calculez la distance totale d'une tournée.** Nous allons maintenant implémenter une méthode qui prend en paramètre une tournée, soit :
  - un vecteur de string correspondant à une combinaison de villes où l'ordre de ces dernières correspond à l'ordre de passage du "voyageur de commerce".

  Il ne faut pas oublier de "boucler la boucle". Ainsi, si "Lille Metz Cannes" est une tournée, la méthode devrait retourner la somme suivante : `DIST[indiceLille][indiceMetz] + DIST[indiceMetz][indiceCannes] + DIST[indiceCannes][indiceLille].`
  Le dernier élément correspond donc au retour à la ville de départ. La signature de cette méthode peut s'inspirer de la suivante :

```
int calculTotalDistanceTournee (
  std::vector<std::string> & vecteurDeNomsDeVille,
  std::vector<std::vector<int> > & DIST,
  std::map<std::string, std::tuple<int, int, int> > &
                                        maMapNomsVillesEtCoordonnees)
```

A vous de réfléchir à comment récupérer l'indice de la ville.

- **C++. Modifiez la méthode `toutesLesPermutations`** de façon à enregistrer dans un tableau `<array>` des `<pair>`es comportant une combinaison (i.e., un vecteur de `<string>`) et les distances totales associées. Vous pouvez baser votre structure de données sur l'exemple suivant :

  ```
  std::array<pair<vector<string>, int>, nombreCombinaisons>
  toutesLesTourneesEtLeurDistanceTotale;
  ```

  Exceptionnellement[10], vous pouvez déclarer `toutesLesTourneesEtLeurDistanceTotale` comme variable globale. Il convient de le faire à la suite des différentes constantes globales. Ainsi, pour chaque combinaison, soit chacune des tournées possibles du voyageur de commerce, **`toutesLesPermutations`** va appeler la méthode calculant la distance totale et va l'enregistrer dans le `<array>`.

[10] Vous avez dû apprendre que les "variables globales" sont à éviter la plupart du temps, on en a déjà utilisé plus tôt pour les constantes. Une des raisons principales est celle du nommage et la difficulté de savoir qui fait quoi lorsque plusieurs variables ont le même nom. Des problèmes de sécurité sont aussi en jeu. Ce sont de bonnes habitudes à prendre, et il faut se projeter sur des projets plus grands où les conflits de nommage peuvent réellement avoir lieu. Ici, on vous permet de facilement modifier les constantes du programme et d'éviter la difficulté d'utiliser des `<array>` assez complexes en paramètre de fonction.

- **C++. Développez un foncteur ou une fonction de comparaison afin de trier les tournées**. Le tableau de paires `toutesLesTourneesEtLeurDistanceTotale` contient désormais une distance totale associée à chaque tournée. Il faut ici trier ce tableau de façon à ce que le premier élément corresponde à la tournée de plus petite distance. Utilisez l'algorithme `sort` dont la documentation est la suivante :
  - http://www.cplusplus.com/reference/algorithm/sort/.
  - Cette documentation vous donne un exemple d'utilisation avec une fonction et avec un foncteur. Il faut voir ici que le foncteur ou la fonction vont être utilisés à chaque comparaison de deux éléments du tableau, opération appliquée à chaque étape du triage. Regardez en particulier le troisième argument de `sort`. Réfléchissez bien à ce qui doit être comparé. Repérez ainsi les valeurs correspondantes aux deux distances totales des deux tournées à comparer. Vous avez d'autres exemples dans le cours.

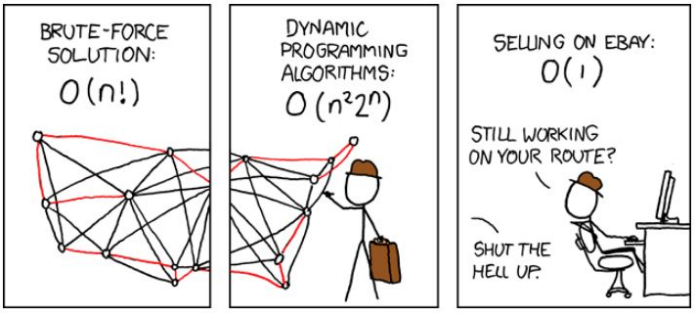
- **C++. Afficher la ou les solutions optimales.** Terminez ce travail en développant une dernière fonction qui va afficher la ou les tournées qui ont la plus petite distance totale. Testez enfin votre travail sur différentes quantités de villes jusqu'à cibler la plus grande quantité permettant d'obtenir une solution en moins d'une minute.

- **Question.** Pour quelles raisons obtient-on systématiquement plusieurs solutions optimales ?

Conclusion. Vous venez de développer une méthode “Brute Force” où toutes les solutions ont été générées. Cette méthode peut être classée dans les méthodes exactes, dans le sens où on obtient la ou les solutions optimales avec certitude, à condition bien sûr de n’avoir aucune restriction de temps. Remarquez l'**explosion combinatoire** liée à l'augmentation du nombre de villes fournies à l’entrée du programme. On peut se référer au tableau ci-dessous, où l’on considère qu’il existe ‘`0.5(n-1)!`‘ solutions. Ainsi, ce type de méthode n’est envisageable que sur de petites valeurs de `n`, on parlera alors d’instance du problème de petite taille. Pour des moyennes et grandes instances, l’utilisation seule des méthodes “Brute Force” est une absurdité à cause de l’explosion combinatoire. Mais il ne faut pas croire pour autant qu’il existe d’autres méthodes qui pourront résoudre des problèmes de très grandes tailles[11]. Ainsi, de nombreux problèmes liés au transport ne disposent pas, encore aujourd’hui, de bonnes solutions (imaginez par exemple la gestion d’un parc de véhicules autonomes à Paris). Mais, ceci ne sera peut-être pas toujours le cas, en particulier grâce à l’**informatique quantique** où d’ailleurs, avant la mesure de l’état des fameux “qubits”, la superposition des états de ces derniers, permet de manipuler d’une certaine manière, un algorithme brute force puisque toutes les combinaisons sont considérées. Ainsi, le domaine de l’optimisation s’offrira une nouvelle jeunesse, si les promesses faites sur ces machines sont respectées.

|Villes|	3	4	5	6	7	8	9	10	15	20
Nombre Solutions	1	3	12	60	360	2520	20160	181440	43 589 145 600	$6{,}082 \times 10^{16}$

Sycamore, un ordinateur quantique de Google



Pic : https://xkcd.com/399/

[11] Mais heureusement il en existe qui permettent de résoudre des instances de taille moyenne.

Bonus.

a. Reprenez le code en s'assurant qu'il ne soit pas possible de générer deux villes dans le même cercle de rayon 1. Vous pouvez encore utiliser les méthodes de `<cmath>` pour vous aider.
b. Établissez un protocole permettant de comparer les deux algorithmes combinatoires vus jusque-là : celui du TP1 et celui donné dans ce document. Discutez des différences principales entre ces deux algorithmes, des résultats attendus puis des résultats obtenus. Vous pouvez à nouveau utiliser `<chrono>`.
c. Mettez vous dans le contexte d'un projet qui devrait durer dans le temps et surtout repris par des collègues. Ainsi, apprenez à vous servir de la documentation Doxygen [12] et la générer pour ce TD&P. Ce bonus qui revient sur plusieurs TD&Ps vous demande d'écrire les différentes balises Doxygen dans vos commentaires avant de générer les fichiers .html de documentation. Vous trouverez plus d'information dans votre cours qui peut être complété de ces deux documentations :
    i. Rapide Aperçu > https://projet.liris.cnrs.fr/mepp/download/Initiation%20%C3%A0%20Doxygen%20pour%20C%20et%20C++.pdf
    ii. Celle des concepteurs > http://www.doxygen.nl/manual/docblocks.html.
d. Reprendre la génération des villes et le calcul des distances avec des coordonnées GPS, i.e., en ajoutant l'altitude comme 3ème coordonnées. En lisant la documentation de la méthode déjà utilisée, vous pourrez rapidement comprendre qu'elle pourra être à nouveau utilisée.
e. Développez deux méthodes factorielles, une itérative et une récursive, puis en utiliser une pour éviter d'utilisation de la constante `constexpr int nombreCombinaisons`.
f. Représentez les différentes villes et une des tournées optimales grâce à GNUPLOT :
    i. (Windows & visual studio) https://marketplace.visualstudio.com/items?itemName=MarioSchwalbe.gnuplot
    ii. (Linux) installation (pour distribution type Debian) :
        1. `sudo apt-get install gnuplot` (distributions basées sur Debian)
        2. `sudo dnf install gnuplot` (Red Hat & Fedora)

Bon courage.

[12] http://www.doxygen.nl/. Quelques vidéos peuvent vous aider : https://www.youtube.com/watch?v=j24jk5zDHJs ; pour l'utiliser directement dans Ecilpse CDT : https://www.youtube.com/watch?v=qYTg_YOiPNA, ou encore une initiation écrite relativement complète : http://www2.ift.ulaval.ca/~eude/Gif-1003/Documents/Initiation-a-Doxygen-pour-C-Cpp.pdf.