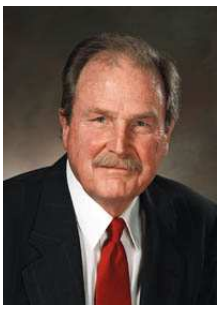
Théorie des codes - TP 1
ZZ3 F5 - Réseaux et Sécurité Informatique
Codage Entropique : Codage de Huffman

Le codage de Huffman est un algorithme de compression de données sans perte. Il a été élaboré par David Albert Huffman, lors de sa thèse de doctorat au MIT et a été publié en 1952 dans l'article *A Method for the Construction of Minimum-Redundancy Codes*, dans les *Proceedings of the Institute of Radio Engineers*.
Le codage de Huffman utilise un code à longueur variable (*Variable-length code* - VLC) pour représenter un symbole de la source. Le code est déterminé à partir d'une estimation des probabilités d'occurrence des symboles : un code court étant associé aux symboles les plus fréquents. Les codes de Huffman sont des codes optimaux, au sens de la plus courte longueur.



D. A. Huffman
(1925-1999)

Exercice 1 Codage de Huffman

1. Réaliser en C++ un algorithme de codage de Huffman en partant du squelette mis à votre disposition (cf. Annexe). Vous utiliserez les fréquences données par défaut dans le code.
2. Pour visualiser l'arbre binaire ainsi obtenu, réaliser un export au format `DOT`.
3. Réaliser ensuite un algorithme de calcul des fréquences des symboles à partir du fichier à coder (`text.txt`).
4. Calculer l'entropie de cette source, ainsi que la longueur moyenne des codes de Huffman.
5. Calculer alors le taux de compression par rapport à un codage ASCII (tenez compte du dictionnaire à envoyer).

Algorithm 1 Codage de Huffman

```
Estimer les probabilités d'occurrences (ou fréquences) des symboles issus de la source ;
Créer une forêt d'arbre binaire avec un seul nœud ;
Chaque nœuds (feuilles) est constitué d'un nom de symbole, d'une fréquence et de son code ;
while Il reste plus qu'un arbre dans la forêt do
    Retirer les deux arbres de la forêt avec les plus petites fréquences dans leurs racines ;
    Créer un nouveau nœud qui sera la racine d'un nouvel arbre ;
    Insérer les 2 arbres retirés dans le nouvel arbre comme des sous-arbres à gauche et à droite ;
    Ce nouveau nœud a une fréquence qui est la somme des fréquences des deux sous-arbres ;
    On rajoute au code du sous-arbre gauche la valeur "0", et "1" dans le sous-arbre droit ;
    Ajouter cet arbre à la forêt ;
end while
return Le seul arbre de la forêt qui est l'arbre de Huffman
```

Exercice 2 Decodage de Huffman

1. Réaliser en C++ un algorithme de decodage de Huffman.

Appendix Priority queues and priority queue implementations

The **Huffman coding algorithm** takes as input the probability of occurrence of each symbol in the alphabet used by an information source, and constructs a tree representing a code for that information source.
The data structure holding the Huffman forest needs to support these operations : create an initially empty structure, insert a tree into the structure and delete from the structure, and return, the tree with the smallest frequency in its root.

We could use an array or a list, but the **priority queue** is faster. These are exactly the typical operations provided by the priority queue abstract data type, used in many fundamental algorithms : create an empty Priority Queue, insert an item into the Priority Queue, delete and return the item in the priority queue is faster.

A **class definition** for a Huffman code tree node :

```
class Symbol {
  public:
    string name;
    double freq;
    string code;
    bool leaf;
    Symbol *left, *right;};
```

The class definition could also have some methods and constructors for commons operations and initialization. When building a new node, the fields of child nodes (left and right) need to be set appropriately. The result is a tree data structure that is useful both for coding and decoding. For coding, you should have an array or other table structure of pointers (`vector<Symbol*>`) to code tree leaf nodes.

A C++ `priority_queue` is a generic container, and can hold any king of things as specified with a template parameter when it is created. However, objects in a priority queue must be comparable to each other for priority. By default, a `priority_queue<T>` use `operator<` defined for objects of type `T`. So, you could just overload that operator for `Symbol`.

```
class Symbol {
  public:
    ...
    bool operator<(Symbol const &) const;
};
```

If you create an STL container such as `priority_queue` to hold `Symbol` objects, then adding an `Symbol` object to the `priority_queue` creates a copy of the `Symbol` object and adds this copy. Dealing with multiple copies of nodes in a tree can be tricky. So you might want to instead, hold pointers to `Symbol`. But there's a problem : our `operator<` is a member function of `Symbol` class. It is not defined for pointers to `Symbol`.
Solution : The template for `priority_queue` takes 3 arguments :

```
template < class T,
           class Container = vector<T>,
           class Compare = less<typename Container::value_type> >
         class priority_queue
```

Compare ⇒ Comparison class ⇒ A class such that the expression comp(a,b), where comp is an object of this class and a and b are elements of the container, returns `true` if a is to be placed earlier than b in a strict weak ordering operation. Here we must define a class implmenting the function call operator `operator()` that performs the required comparison. Now we can create the priority queue as :

```
std::priority_queue<Symbol*, std::vector<Symbol*>, myClassPtrComp > pq;
```