Un bref aperçu sur le cours

1. Notion d'algorithme et d'espace mémoire

L'objectif initial de l'informatique était de fournir des *algorithmes* permettant de résoudre *un problème* donné.

- Algorithme : ensemble de règles opératoires dont l'application permet de résoudre le problème au moyen d'un nombre fini d'opérations:

  → problème : temps d'exécution des traitements *à minimiser*

- Espace mémoire : taille des données utilisées dans le traitement et la représentation du problem.

  → problème : espace mémoire occupé *à minimiser*

Les notions de *traitements* et *d'espace mémoire* sont liés. Ces deux critères doivent servir de guide au choix d'une représentation de donnée.

2. Approche intuitive de la complexité

Exemple de problème: le voyageur de commerce.

Ce voyageur doit trouver le chemin le plus court pour visiter tous ses clients, localisés chacun dans une ville différente, et en passant une et une seule fois par chaque ville.

On peut imaginer un algorithme naïf :

1. on envisage tous les chemins possibles,
2. on calcule la longueur pour chacun d'eux,
3. on conserve celui donnant la plus petite longueur.

Cette méthode « brutale » conduit à calculer un nombre de chemin «exponentiel» par rapport au nombre de villes. Il existe des algoritmes plus « fins » permettant de réduire le nombre de chemins à calculer et l'espace mémoire utilisé.
Mais on ne connaît pas d'algorithme exact qui ait une complexité (temps de résolution) acceptable pour ce problème.

Pour les problèmes qui sont aussi «difficile», on cherche des méthodes approchées, appelées *heuristiques.*

3 Complexité en temps (et mémoire)

Si l'on souhaite comparer les *performances* d'algorithmes, on peut considérer une mesure basée sur leur *temps d'exécution*. Cette mesure est appelée ***la complexité en temps de l'algorithme***.

On utilise la notion dite « de Landau » qui traite de l'ordre de grandeur du *nombre d'opérations* effectuées par un algorithme donné.

→ on utilise la notation « **O** » qui donne *une majoration de l'ordre de grandeur du nombre d'opérations.*

Pour déterminer cette majoration, il faudra :

- Connaître la **taille n de la donnée** en entrée du problème (exemple: nombre de données à traiter, le degré d'un polynôme, taille d'un fichier, le codage d'un entier, le nombre de sommets d'un graphe, etc.)
- Déterminer les *opérations fondamentales* qui interviennent dans ces algorithmes et qui sont telles que les temps d'exécution seront directement proportionnels au nombre de ces opérations.

Exemple : calculer la puissance n-ième d'un entier (n étant positif) en C :

```
int puissance (int a, int n)
{
        int i, x ;

        x=1 ;
        for (i = 0; i < n; i=i+1)       // boucle de n iterations
                x = x * a ;             // opération fondamentale de multiplication

        return x ;
}
```

La complexité de cet algorithme est en temps linéaire O(n)

Remarque 1 : les performances de la machine n'interviennent pas directement dans l'ordre de grandeur de la complexité.

Remarque 2 : la théorie de la complexité a pour but de donner un contenu formel à la notion intuitive de *difficulté de résolution d'un problème.*

Type de complexité algorithmique

On considère désormais un algorithme dont le temps maximal d'exécution pour une donnée de taille n en entrée est noté T(n).

Chercher la *complexité au pire* – dans la situation la plus défavorable – c'est exactement exprimer T(n) en général en notation O. Par exemple :

- T(n) = O(1), *temps constant* : temps d'exécution indépendant de la taille des données à traiter.
- T(n) = O(log(n)), *temps logarithmique* : on rencontre généralement une telle complexité lorsque l'algorithme casse un gros problème en plusieurs petits, de sorte que la résolution d'un seul de ces problèmes conduit à la solution du problème initial.
- T(n) = O(n), *temps linéaire* : cette complexité est généralement obtenue lorsqu'un travail en temps constant est effectué sur chaque donnée en entrée.
- T(n) = O(n.log(n)) : l'algorithme scinde le problème en plusieurs sous-problèmes plus petits qui sont résolus de manière indépendante. La résolution de l'ensemble de ces problèmes plus petits apporte la solution du problème initial.
- T(n) = O(n^2), *temps quadratique* : apparaît notamment lorsque l'algorithme envisage toutes les paires de données parmi les n entrées (ex. deux boucles imbriquées)
  remarque : O(n^3) *temps cubique*
- T(n) = O(2^n), *temps exponentiel* : souvent le résultat de recherche brutale d'une solution.

On distingue également d'autres type de complexité: la compelxité en moyenne et la compelxité amortie.

La complexité en moyenne nécessite une connaissance de la distribution probabliliste des données. La compelxité amortie mesure quant à elle mesure la complexité moyenne d'une opération quand celle-ci est exécutée plusieurs fois de suite. C'est en somme la complexité dans le pire cas d'une opération quand elle est exécutée plusieurs fois. Cette mesure prend en compte le fait que les données peuvent changer de positions `lors de l'exécution d'une opération plusieurs fois.

Complexité en place mémoire

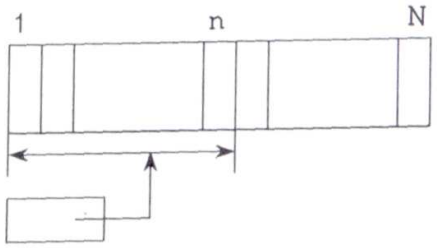
L'analyse de la complexité en place mémoire revient à évaluer, en fonction de la taille de la donnée, la place mémoire nécessaire pour l'exécution de l'algorithme, en dehors de l'espace occupé par les instructions du programme et des données.

Conclusion : Il s'agit donc d'obtenir le meilleur compromis espace-temps.

4. Structure de données et complexité

Implantation contiguë de liste (tableau)

La première représentation d'un objet de type liste consiste à prendre une zone de mémoire contiguë, et à y mettre les contenus des places successives de la liste.



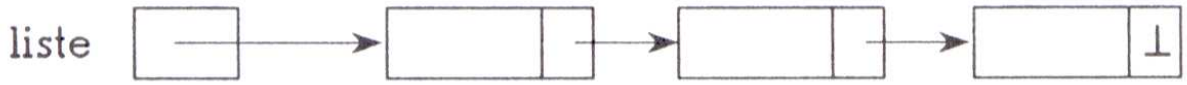
On peut constater que cette représentation permet une implémentation simple :

- complexité constante (indépendante de la longueur) des opérations d'accès : v=L[i], v prend la valeur du i-ème élement du tableau L
- l'insertion d'un élément à la k-ième place entraîne un déplacement de tous les éléments situés derrière lui, donc N-k+1 déplacements.
- suppression d'un élément situé à la k-ième place entraîne le déplacement de tous les éléments situés derrière lui, donc N-k déplacements.

La complexité des opérations d'insertion et de suppression est donc au mieux (cas de la fin de la liste) en $\theta(l)$, au pire (cas du début de liste) en $\theta(n)$, et en moyenne en $\theta(n/2)$.

Implantation chaînée des listes

Une liste chaînée consiste à lier entre elles des zones de mémoire, chacune d'elles contenant la représentation d'un élément de la liste, et un pointeur vers la suivante.



On peut constater que cette représentation permet une implantation simple des opérations premier, successeur, insérer, supprimer un élément : θ (l)
Par contre les accès par le rang ne peuvent plus être obtenus que par un parcours séquentiel, donc ont une complexité au mieux (début de liste) en θ (l), au pire (fin de liste) en θ(n), en moyenne en θ (n/2).

Remarques :

- Une liste est une structure orientée vers les traitements séquentiels.

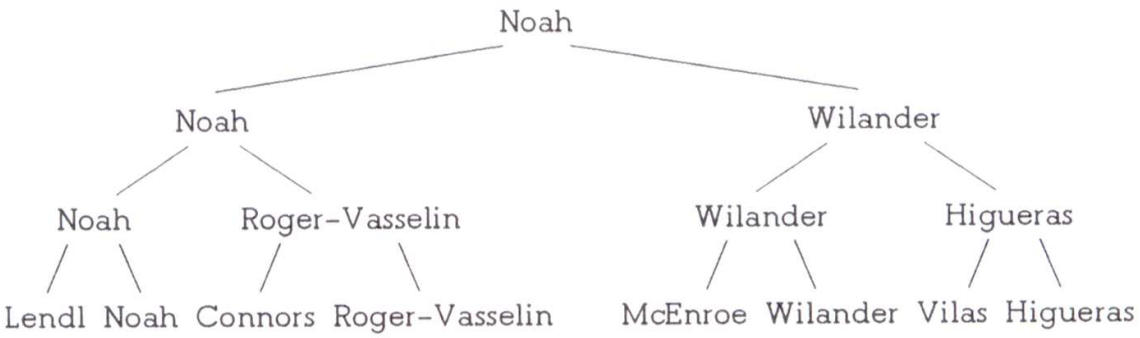
- Les piles (LILO) et les files (FIFO) sont des cas particuliers de liste qui ont des représentations contiguës ou chaînées efficaces.

- Les fichiers séquentiels sont une certaine forme de liste.

Structures arborescente

Les structures arborescentes sont les structures de données les plus importantes en informatique.

La notion d'arbre est une notion utilisée dans la vie courante, ailleurs qu'en informatique : Les résultats d'un tournoi de tennis



Toute recherche d'un élément par comparaison à l'élément racine sera proportionnelle à la hauteur de l'arbre O(log (n)).

Exemple : Est-ce que Noah a battu McEnroe ?

Recherche dans une liste quelconque

L'algorithme de recherche d'un élément dans une liste quelconque a une complexité proportionnelle au rang de l'élément recherché.

On peut en conclure que la recherche est alors au mieux en $\theta(l)$, en moyenne en $\theta(n/2)$, et au pire en $\theta(n)$.

Arbres de recherche

Si on peut *ordonner une liste*, il est possible d'utiliser la recherche dichotomique. La recherche est alors en $\theta(\log_2 n)$, les adjonctions et suppressions sont en $\theta(n)$, ce qui est un frein important à cette représentation.

Le principe même de la dichotomie permet la structuration en **arbre binaire** de l'ensemble des éléments : l'élément de rang (i+j)/2 est placé à la racine, ceux qui le précèdent sont mis dans le sous-arbre gauche, et ceux qui le suivent dans celui de droite. On répète recursivement cette opération sur chacun des sous-arbres.

De tels arbres sont appelés ***arbres binaires de recherche***, et ils conduisent à des opérations simples et efficaces (complexité en temps logarithmique en moyenne, linéaire dans le pire).

Arbres H-équilibrés

Les opérations sur les arbres binaires sont d'une complexité moyenne $\theta(\log n)$, mais il peut exister des arbres pour lesquels la complexité est en $\theta(n)$ (peigne).

Il peut arriver que l'on ne puisse accepter les cas défavorables. Comme ces cas sont dus à de grandes variations dans les longueurs des branches de l'arbre, on peut essayer d'éviter ces variations.

Il est possible de construire des arbres de recherche tels que tous les niveaux soient complets mais cela nécessite en général des réorganisations trop coûteuses lors des adjonctions et des suppressions pour maintenir cette propriété.

Dans les années 1960, Adelson-Velskii et Landis ont introduit une classe d'arbres équilibrés qui se rapprochent de cette propriété (només AVL).
L'idée est que pour tout nœud d'un AVL, la différence de hauteur entre ses sous-arbres gauche et droit est au plus de 1.
La complexité de l'adjonction est au pire égale à la hauteur de l'arbre, en terme de comparaisons, ou de modifications du déséquilibre.

Arbres balancés

Lorsque le nombre d'éléments d'un ensemble devient important, la représentation ne peut plus être conservée en mémoire centrale.

La taille de l'espace mémoire nécessaire à la représentation d'un nœud d'un arbre binaire de recherche est en général inférieure à la taille d'un secteur de disque.

Aussi est-on amené à mettre plusieurs nœuds par secteur ou bloc disque (groupe de secteurs consécutifs transférés en une seule opération).

Si les nœuds de l'arbre, même H-équilibré, sont répartis n'importe comment sur ces blocs, le parcours d'une branche de l'arbre demandera autant d'accès disque qu'il y a de nœuds sur cette branche.

Ce nombre n'est pas très important, puisqu'il est en $\log_2 n$, mais le temps d'accès disque étant de l'ordre de 50 ms, ceci peut conduire à des temps de recherche dépassant la seconde.

Pour réduire le nombre d'accès disque, on peut essayer de faire en sorte que les nœuds voisins sur l'arbre se retrouvent dans un même bloc, permettant de parcourir plusieurs nœuds sans avoir d'accès disque.

L'autre solution est d'augmenter la taille des nœuds en mettant plus de valeurs dans un nœud.

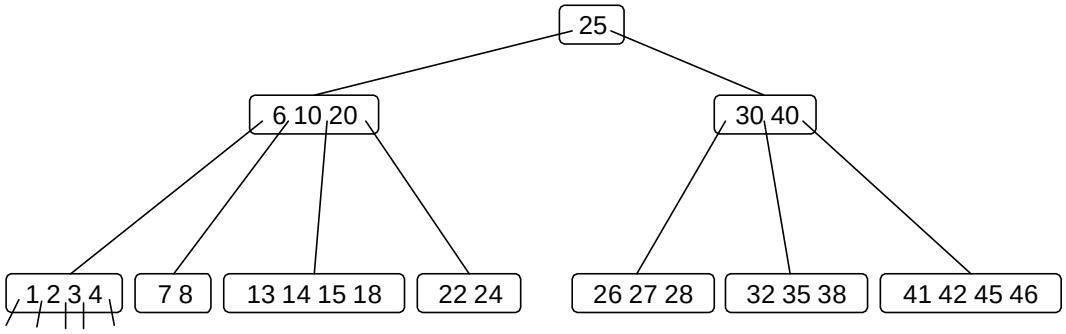
Ceci conduit à généraliser la notion d'arbre de recherche aux arbres généraux et forêts. Un arbre balancé est appelé B-arbre.

Comme le degré des B-arbres est borné, les opérations de recherche, adjonction et suppression ont une complexité, en terme d'opérations de traitement d'un nœud, proportionnelle à la hauteur des arbres.

Diminuer la hauteur de l'arbre a donc essentiellement pour but de réduire le nombre des accès disque, dont on sait qu'ils sont près de 10 000 fois plus lents

que les accès mémoire centrale. Même si le traitement d'un nœud est en θ(m), il restera inférieur au temps d'accès disque.

B-arbre d'ordre 2



Conclusion

- Un algorithme peut avoir une complexité faible, mais dans la pratique présenter des temps d'exécutions prohibitifs.

- La complexité d'un algorithme est toujours relative à une ou plusieurs opérations fondamentales qu'il faut préciser au préalable. Elle dépend de la taille des données, et pour une taille donnée, de la configuration de ces données.

- Les seuls algorithmes pratiquement utilisables sont ceux dont la complexité est inférieure à n^2. Entre n^2 et n^3 on ne peut traiter que des problèmes de taille moyenne. Au-delà, on ne pourra traiter que des problèmes de très petite taille.

- L'amélioration des performances des machines ne change pas fondamentalement l'efficacité d'un algorithme.

Référence : ″Types de données et Algorithmes″ C. Froidevaux, M.C. Gaudel, M. Soria, Ediscience international.