Complexité et algorithmique avancé

Cours du Master « Sécurité » 1ère année

Chargé de cours: H. Mosteghanemi
Département d'Informatique
Faculté d'Electronique et d'Informatique
Université des Sciences et de la Technologie Houari Boumediène
BP 32, El-Alia, Bab Ezzouar
DZ-16111 ALGER
Email: hmosteghanemi@usthb.dz

Chapitres clés du cours

- Chapitre introductif
- Les bases de l'analyse d'un algorithme
- Quelques outils mathématiques
- Structure de données avancées
  - Arbre et graphe
  - Dictionnaire, index et technique de hachage
- Analyse d'algorithme de tri
- Les classes de problèmes
- Stratégies de résolutions des problèmes
- Algorithmique du texte

Chapitre 1 : Introduction

- En informatique, deux questions fondamentale se posent toujours devant un problème à résoudre :
  - Existe-t-il un algorithme pour résoudre le problème en question ?
  - Si oui, cet algorithme est-il utilisable en pratique, autrement dit le temps et l'espace mémoire qu'il exige pour son exécution sont-ils « raisonnables »

La première question traite de la ***calculabilité*** et la seconde traite de la ***complexité***

Chapitre 2 : Les bases de l'analyse d'un algorithme

- Notion de problème

Un problème est une question qui a les propriétés suivantes :

1. Elle est générique et s'applique à un ensemble d'éléments;
2. Toute question posée pour chaque élément admet une réponse

Chapitre 2 : Les bases de l'analyse d'un algorithme

- ***<u>Notion d'instance</u>***

Une instance de problème est la question générique appliquée à un élément.

- ***<u>Exemple :</u>***
  - L'entier 15 est-il pair ou impair est une instance du problème de parité des nombres

Chapitre 2 : Les bases de l'analyse d'un algorithme

- ***<u>Notion de solution</u>***

La solution d'un problème donné est un objet qu'il faut déterminer et qui satisfait les contraintes explicite imposées dans le problème. Il existe quatre manières d'obtenir la solution d'un problème donné :

1. Appliquer une formule explicite
2. Utiliser une définition récursive
3. Utiliser un algorithme qui converge vers la solution
4. Énumérer les cas possibles.

Chapitre 2 : Les bases de l'analyse d'un algorithme

- ***<u>Notion de solution</u>***

La solution d'un problème donné est un objet qu'il faut déterminer et qui satisfait les contraintes explicite imposées dans le problème. Il existe quatre manières d'obtenir la solution d'un problème donné :

1. Appliquer une formule explicite
2. Utiliser une définition récursive
3. Utiliser un algorithme qui converge vers la solution
4. Énumérer les cas possibles.

Chapitre 2 : Les bases de l'analyse d'un algorithme

- ***<u>Notion de programme</u>***
  - Un processus de solution d'un problème pouvant être exécutée par un ordinateur.
  - Il contient toute l'information nécessaire pour résoudre le problème donné en un **<u>temps fini</u>**

Chapitre 2 : Les bases de l'analyse d'un algorithme

- ***<u>L'analyse d'un algorithme</u>***
  - L'analyse des algorithmes s'exprime en termes d'évaluation de la complexité de ces algorithmes.
  - L'analyse d'un algorithme produit également :
    - La modularité
    - La correction
    - La maintenance
    - La simplicité
    - La robustesse
    - L'extensibilité
    - Le temps de mise en œuvre, … etc.

Chapitre 2 : Les bases de l'analyse d'un algorithme

- ***<u>Élément de base de la complexité</u>***

<u>Définition 1:</u> un algorithme est un ensemble fini d'instructions qui, lors de leur exécution, accomplissent une tâche donnée.

<u>Définition 2:</u> Un algorithme est un ensemble d'opérations de calcul élémentaires, organisé selon des règles précises dans le but de résoudre un problème donné.

- Un algorithme doit satisfaire ces propriétés :
  - Il peut avoir zéro ou plusieurs quantités de données en entrée;
  - Il doit produire au moins une quantité en sortie;
  - Chacune de ses instructions doit être claire et non ambiguë;
  - Il doit se terminer après un nombre fini d'étapes;
  - Chaque instruction doit être implémentable;

Chapitre 2 : Les bases de l'analyse d'un algorithme

- Exemple : Soit le problème « trier un ensemble de n nombres entiers n>1 ».

Une solution pourrait être :

« Rechercher le minimum parmi les entiers non triés et le placer comme successeurs de la succession trié » cette expression ne constitue pas un algorithme pour résoudre le problème

??? Proposez un Algorithme.

Chapitre 2 : Les bases de l'analyse d'un algorithme

<u>Qu'est ce que l'analyse ?</u>

- Déterminer la quantité des ressources nécessaires à son exécution
- Ces ressources peuvent être :
  - La quantité de mémoire utilisée;
  - Le temps de calcul;
  - La largeur de la bande passante, .. Etc.
- Nous nous intéressons dans ce cours au temps et à la mémoire, le but est d'identifier face à plusieurs algorithmes qui résolvent un même problème, ***celui qui est le plus efficace***.

Chapitre 2 : Les bases de l'analyse d'un algorithme

<u>Algorithme et Complexité :</u>

- La complexité d'un algorithme est la mesure du nombre d'opérations élémentaires qu'il effectue pour le problème pour lequel il a été conçu
- La complexité est mesurée en fonction de la taille du problème ; la taille étant elle-même mesurée en fonction des données du problème
- La complexité est par conséquent mesurée en fonction des données du problème

Il s'agit donc de trouver une équation qui relie le temps d'exécution à la taille des données.

Chapitre 2 : Les bases de l'analyse d'un algorithme

Trois Types de complexité

- Complexité au meilleur cas : cette mesure à peu d'intérêt et ne permet pas de distinguer deux solutions.
- Complexité en moyenne : nécessite la connaissance de la distribution des données.
- Complexité du pire cas : Fournit une borne supérieure du temps d'exécution qui indique l'algorithme ne peut pas dépasser cette valeur. C'est cette complexité qui nous intéresse dans la suite de ce cours.

Chapitre 2 : Les bases de l’analyse d’un algorithme

Paramètres de l’analyse

- Les performance d’un algorithme dépendent des trois principaux paramètres suivants :
  - La taille des entrées : quelque soit le problème, il faut en premier lieu caractériser les entrées.
  - La distribution des données
    - Exemple : données triés ou non
  - La structure de données

Chapitre 2 : Les bases de l'analyse d'un algorithme

La complexité asymptotique

- En pratique, il est difficile de calculer de manière exacte la complexité d'un algorithme.
- La complexité asymptotique est une approximation du nombre d'opération que l'algorithme exécute en fonction de la donnée en entrée (donnée de grande taille) et ne retient que le terme de poids fort dans la formule et ignore les coefficient multiplicateur.
- La complexité d'un algorithme est indépendante du type de machine sur laquelle il s'exécute.

Chapitre 2 : Les bases de l'analyse d'un algorithme

Exemple: Calcul du maximum de quatre valeurs a, b, c et d.

- Proposez deux solutions différentes
- Proposez une généralisation de l'algorithme au calcul du maximum d'une suite.

Chapitre 2 : Les bases de l'analyse d'un algorithme

```
Début
   Max = a;
   Si (a > max) alors max = b; fsi;
   Si (c > max) alors max = c; fsi;
   Si (d > max) alors max = d; fsi;
 Fin.
```

```
Début
Si (a > b) alors
          Si (a > c) alors
                     Si (a > d) alors max = a;
                         sinon max = d; fsi;
                     sinon
                     Si (c > d) alors max = c;
                          sinon max = d; fsi;
                     Fsi;
          Sinon
          Si (b > c) alors
                     Si (b > d) alors max = b;
                         sinon max = d; fsi;
                     Sinon
                     Si (c > d) alors max = c;
                          sinon max = d; fsi;
Fsi;      Fsi;
Fin.
```

Chapitre 2 : Les bases de l'analyse d'un algorithme

La notation de Landau :

- On ne mesure généralement pas la complexité exacte d'un algorithme
- On mesure son ordre de grandeur qui reflète son comportement asymptotique, c'est-a-dire son comportement sur les instances de grande taille

Chapitre 2 : Les bases de l'analyse d'un algorithme

La notation de Landau :

- f = **O(g)** ssi il existe n_0, il existe c≥0,

  pour tout n ≥ n_0, f(n) ≤ c*g(n)
- f = **Ω(g)** ssi g = O(f)
- f = **o(g)** ssi pour tout c≥0, il existe n_0,

  pour tout n ≥ n_0, f(n) ≥ c*g(n)
- f = **Θ(g)** ssi f = O(g) et g = O(f)

Autrement dit :

$$\exists\, C_1, C_2 > 0, \exists\, n_0 \geq 0,$$
$$0 \leq C_1 \times g(n) \leq f(n) \leq C_2 \times g(n).$$

Chapitre 2 : Les bases de l'analyse d'un algorithme

<u>*Synthèse :*</u>

- Définition de problème, d'instance de solution et d'algorithme
- Le calcul de la complexité est l'objectif de base de l'analyse d'un algorithme
- Déterminer la complexité revient à trouver une formule qui relie le temps d'exécution à la taille du problème.
- Il existe trois types de complexité mais c'est la complexité au pire qui est la plus significative.
- La complexité des opérations de base est de O(1)
  - Lecture/écriture, affectation, opérations arithmétiques.
- Il n'existe pas de méthode automatique
- L'analyse fait appel à des outils mathématiques
  - L'algèbre, la théorie élémentaire des probabilités … etc.

Chapitre 3: Outils mathématiques

- Sommation
- Série et suites numériques
- Récurrences
- Calcul des sommes par intégrales
- Résolution des équations de récurrence

Chapitre 3: Outils mathématiques

- Calculer la complexité d'un algorithme revient à déterminer une formule mathématique qui exprime le coût de cet algorithme.
- Cette formule s'obtient en bornant des expressions de sommations et/ou de produits par un calcul de limite à l'infini
- Dans plusieurs situations les propriétés relationnelles entre les nombres réels s'appliquent aussi aux fonctions
- Ce chapitre donne quelques rappels fondamentaux liés à l'utilisation des fonctions

Chapitre 3: Outils mathématiques

- ***Sommation:***
  - Lorsqu'un algorithme contient une structure de contrôle répétitive, le temps d'exécution s'exprime comme une somme des temps pour exécuter chaque instruction dans le corps de la boucle.
  - Déterminer une fonction de complexité asymptotique exige le calcul d'expressions de sommation de série et savoir les bornées.

Chapitre 3: Outils mathématiques

- ***<u>Formules et propriétés des Sommations:</u>***

Soit une séquence de nombres $a_1, a_2, a_3, \ldots, a_n$

La somme infinie $a_1+a_2+\ldots+a_n+\ldots$

$$\sum_{k=1}^{\infty} a_k \quad \text{Est interprétée par :} \quad \lim_{n \longrightarrow \infty} \sum_{k=1}^{k=n} a_k$$

- Si la limite n'existe pas, la série diverge, autrement elle converge
- Si n n'est pas entier on suppose que la limite supérieur est $\lfloor n \rfloor$
- Si la somme commence avec $k = x$ et x n'est pas entier on supposera que la sommation commence avec l'entier $\lfloor x \rfloor$

Chapitre 3: Outils mathématiques

- ***<u>Formules et propriétés des Sommations:</u>***

<u>Linéarité</u>

Soient $a_1, a_2, \ldots, a_n$ et $b_1, b_2, \ldots, b_n$ deux séquences de nombres et c un réel, la linéarité signifie :

$$\sum_{k=1}^{k=n}(c * a_k + b_k) = c * \sum_{k=1}^{k=n} a_k + \sum_{k=1}^{k=n} b_k$$

Exemple :

$$\sum_{k=1}^{k=n}(\Theta(f(k)) = \Theta(\sum_{k=1}^{k=n} f(k))$$

Chapitre 3: Outils mathématiques

- ***Séries arithmétiques:***

$$\sum_{k=1}^{k=n} k = 1 + 2 + 3 + \cdots + n = \frac{n * (n+1)}{2} = \Theta(n^2)$$

- ***Séries géométriques:*** Si x est un réel, $x \neq 1$

$$\sum_{k=0}^{k=n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{(x^{n+1} - 1)}{x - 1}$$

Est une série exponentielle (géométrique), si $|x|<1$ et la sommation est infinie, on obtient la série géométrique décroissante suivante:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \qquad \ldots\ (\#)$$

Chapitre 3: Outils mathématiques

- ***Séries harmoniques:*** n>0, le n[ième] nombre harmonique s'écrit:

$$
\begin{aligned}
H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \\
&= \sum_{k=1}^{k=n} \frac{1}{k} \\
&= \ln n + O(1).
\end{aligned}
$$

- ***Séries emboitées:***

$a_0, a_1, \ldots, a_n,$ on écrit :

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0.$$

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n.$$

Chapitre 3: Outils mathématiques

- ***Séries emboitées:***

Exemple : $$\sum_{k=1}^{n-1} \frac{1}{k*(k+1)}$$

On réécrit chaque terme sous la forme :

$$\frac{1}{k*(k+1)} = \frac{1}{k} - \frac{1}{k+1}$$

D'où :

$$\begin{aligned}\sum_{k=1}^{n-1} \frac{1}{k*(k+1)} &= \sum_{k=1}^{n-1} (\frac{1}{k} - \frac{1}{k+1}) \\ &= a_0 - a_n \\ &= 1 - \frac{1}{n}\end{aligned}$$

Chapitre 3: Outils mathématiques

- ***Les produits***

Le produit fini $a_1 * a_2 * \ldots * a_n$ s'écrit :

$$\prod_{k=1}^{k=n} a_k$$

- Si n = 0 le produit vaut 1 par définition
- Une formule de produit peut être convertie en une formule de sommation en passant par le logarithme :

$$\log(\prod_{k=1}^{k=n} a_k) = \sum_{k=1}^{k=n} \log a_k.$$

Chapitre 3: Outils mathématiques

- ***<u>Les produits</u>***

Exercice:

1) Trouver une formule simple pour :

$$\sum_{k=1}^{k=n}(2*k-1).$$

2) En utilisant les séries harmoniques montrer que :

$$\sum_{k=1}^{k=n}\frac{1}{2*k-1}=\ln(\sqrt{n})+O(1)$$

3) En utilisant la série (#), calculer la somme suivante :

$$\sum_{k=0}^{k=n}k*x^k.$$

Chapitre 3: Outils mathématiques

- ***<u>Récurrences</u>***
  - ***<u>Borner une sommation</u>***
- La récurrence est une des techniques que l'on peut utiliser pour borner une sommation
- Par exemple on peut démontrer par récurrence que $\sum_{k=0}^{k=n} k = \frac{n*(n+1)}{2}$
- La récurrence peut aussi être utiliser pour prouver l'existence d'une borne.

<u>Exemple :</u>

Prouver que $\sum_{k=0}^{k=n} 3^k$ est en $O(3^n)$

Chapitre 3: Outils mathématiques

- ***Récurrences***
  - ***Borner une sommation***

Démonstration :

On montre par récurrence, qu'il existe une constante c et un entier n_0 telle que :

$$\sum_{k=0}^{k=n} 3^k \leq c * 3^n, \forall n \geq n_0.$$

Cas de base : n=0, on a : $\sum_{k=0}^{k=n} 3^k = 3^0 = 1 \leq c * 1$pour c ≤1.

Hypothèse de récurrence : on suppose la formule vraie à l'ordre n et on montre qu'elle reste vraie à l'ordre n+1.

$$\sum_{k=0}^{k=n+1} 3^k = \sum_{k=0}^{k=n} 3^k + 3^{n+1} \leq c * 3^n + 3^{n+1}$$

$Or\ c * 3^n + 3^{n+1} = c * 3^{n+1}\left(\frac{1}{3} + \frac{1}{c}\right) \leq c * 3^{n+1}\ si\ \frac{1}{3} + \frac{1}{c} \leq 1\ ce\ qui\ est\ vrai\ si\ c \geq \frac{3}{2}.\ C.Q.F.D$

Chapitre 3: Outils mathématiques

- ***Récurrences*** Attention aux erreurs !
- Dans la définition de O les constantes c et n_0 ne doivent pas varier en n

Contre-exemple : l'affirmation suivante : $\sum_{k=1}^{k=n} k = O(n)$ *est fausse. La démonstration par récurrence suivante est fausse :*

- **cas de base** *: pour n=0 on a :* $\sum_{k=1}^{k=1} k = 1 \leq c * 1$ *vraie pour* $c \geq 1$
- *hypothèse de récurrence :* $\exists c, n_0$ *telles que* $\sum_{k=1}^{k=n} k \leq c * n, \forall n \geq n_0$
- *Calculons* $\sum_{k=1}^{k=n+1} k = \sum_{k=1}^{k=n} k + (n+1) \leq c * n + (n+1) = n * (c+1) + 1$. *La constante c croit avec n.*
- *Calculons* $\sum_{k=1}^{k=n+2} k = \sum_{k=1}^{k=n} k + (n+1) + (n+2) \leq c * n + (n+1) + (n+2) = n * (c+2) + 3$, *on voit bien que la constante c n'est pas unique pour tout* $n > n_0$.

- ***Récurrences***
  - ***Borner un terme par le plus grand terme de la série***

On peut borner une sommation en bornant chacun de ses termes par une borne supérieure intéressante. Il suffit dans plusieurs cas de borner tous les termes par le terme le plus grand, autrement dit :

$$\sum_{k=1}^{k=n} a_k \leq n * max\{a_1, a_2, \ldots, a_n\}$$

Exemple :

$$n! = 1 \times 2 \times 3 \times \cdots \times n \leq n \times n \times n \times \cdots \times n = n^n$$

$$\sum_{k=1}^{k=n} k \leq \sum_{k=1}^{k=n} n = n^2$$

Chapitre 3: Outils mathématiques

- ***Récurrences***
  - ***Borner une série par une série géométrique***

Pour borner une série $\sum_{k=1}^{k=n} a_k$, par une série géométrique, on doit montrer qu'il existe une constante r, telle que r < 1 et $a_{k+1}/a_k \leq 1$, si un tel r existe alors $a_k \leq a_0 * r^k$, $\forall k \geq 0$.

On obtient ainsi :

$$\sum_{k=0}^{k=n} a_k \leq \sum_{k=0}^{\infty} a_0 * r^k =$$

$$a_0 * \sum_{k=0}^{\infty} r^k =$$

$$a_0 * \frac{1}{1-r}.$$

Cette technique permet d'obtenir une meilleure borne que celle qui utilise le plus grand terme.

Chapitre 3: Outils mathématiques

- ***Récurrences***
  - ***Borner une série par une série géométrique***

Exemple: Trouver une borne pour $\sum_{k=1}^{\infty} k/3^k$

Pour k = 1, $a_1 = 1/3$

- Le rapport entre deux termes quelconques vaut :

$$a_{k+1}/a_k = \frac{(k+1)/3^{k+1}}{k/3^k}$$
$$\frac{1}{3} * \frac{k+1}{k}$$
$$\frac{1}{3} * (1 + \frac{1}{k}) \leq \frac{2}{3} \qquad \forall k \geq 1.$$

Tout terme a_k est donc borné par $a_0 * r^k = (1/3)*(2/3)^k$, d'où:

$$\sum_{k=1}^{\infty} \frac{k}{3^k} \leq \sum_{k=1}^{\infty} \left(\frac{1}{3}\right) * \left(\frac{2}{3}\right)^k = \frac{1}{3} * \frac{1}{1 - 2/3} = 1$$

Chapitre 3: Outils mathématiques

- ***Récurrences***
  - ***Borner une série par une série géométrique***

Remarque:

Pour appliquer cette méthode il est nécessaire de trouver une constante r < 1. Si r varie la méthode n'est pas applicable, comme pour la série divergente suivante:

$$\sum_{k=1}^{\infty} 1/k = \lim_{n \longleftarrow \infty} \sum_{k=1}^{n} 1/k = \lim_{n \longleftarrow \infty} \Theta(\ln n) = \infty$$

Dans ce cas le rapport $a_{k+1}/a_k < r$ mais r est une variable qui tend vers 1. Il n'y a pas de r constant, r < 1.

Mais on peut appliquer cette méthode à partir d'un certain rang, à partir duquel il existe un r constant qui vérifie les conditions d'existence d'une série géométrique.

- ***Récurrences***
  - ***Borner une série par une série géométrique***

Exemple:

Par exemple, pour borner la série $\sum_{k=1}^{\infty} \frac{k^2}{2^k}$*, on calcule le rapport* a_{k+1}/a_k. On peut montrer que $a_{k+1}/a_k \leq 8/9$ si $k \geq 3$.

$$\sum_{k=1}^{\infty} \frac{k^2}{2^k} = \sum_{k=1}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k}$$

$$\leq O(1) + 9/8 \sum_{k=3}^{\infty} (8/9)^k$$

$$O(1).$$

Chapitre 3: Outils mathématiques

- ***Calcul des sommes par intégrales***
  - Une fonction est monotone croissante si

$$m \leq n \Rightarrow f(m) \leq f(n)$$

  - Une fonction est monotone décroissante si

$$m \leq n \Rightarrow f(m) \geq f(n)$$

Soit à borner cette expression de sommation :
$\sum_{k=m}^{n} f(k)$

  - Si *f est monotone croissante on à :*

$$\int_{m-1}^{n} f(x)dx \leq \sum_{m}^{n} f(k) \leq \int_{m}^{n+1} f(x)dx$$

  - *Si f est monotone décroissante on à:*

$$\int_{m}^{n+1} f(x)dx \leq \sum_{m}^{n} f(k) \leq \int_{m-1}^{n} f(x)dx$$

Chapitre 3: Outils mathématiques

- ***Calcul des sommes par intégrales***

Exemple:

Bornons $\sum_{k=m}^{n} 1/k$; *par cette méthode,*

Puisque f(k) = 1/k est une fonction monotone décroissante, alors :

$$\int_1^{n+1} \frac{1}{x} dx \leq \sum_1^n \frac{1}{k} \leq \int_0^n \frac{1}{x} dx$$

Nous avons alors d'une part :

$$\sum_{k=1}^{n} \frac{1}{k} \geq \int_1^{n+1} \frac{dx}{x} \geq \ln(n+1).$$

Et d'autre part :

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \sum_{k=2}^{n} \frac{1}{k} \leq 1 + \int_1^n \frac{dx}{x} \leq 1 + \ln n$$

Chapitre 3: Outils mathématiques

- ***<u>Fonctions classiques</u>***
  - ***<u>La partie entière</u>***

C'est une fonction monotone croissante, elle est définie comme suit pour tout réel x :

1. Le plus grand entier inférieur ou égal à x et s'écrit $\lfloor x \rfloor$
2. Le plus petit entier supérieur à x et s'écrit $\lceil x \rceil$.

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

Pour tout entier n :

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$$

Chapitre 3: Outils mathématiques

- ***Fonctions classiques***
  - ***La fonction exponentielle***
- Pour tout réel $a \neq 0$, m, n, on a les identités suivantes :

$$a^0 = 1$$
$$a^1 = a$$
$$a^{-1} = 1/a$$
$$(a^m)^n = (a^n)^m = a^{mn}$$
$$a^m * a^n = a^{m+n}$$

La fonction a^n est monotone et croissante pour $a \geq 1$.

On pose $0^0=1$ si nécessaire.

Comparaison avec un polynôme: $\lim_{n \longrightarrow \infty} \frac{n^b}{a^n} = 0$

Chapitre 3: Outils mathématiques

- ***Fonctions classiques***
  - ***La fonction exponentielle (suite)***
- Une fonction exponentielle avec une base strictement plus grande que 1 croit plus vite qu'un polynôme quelconque. D'où: $n^b = O(a^n)$
- Si on s'intéresse à e = 2.71828… qui est la base qui logarithme népérien on a :

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

D'où $e^x \geq 1 + x$. L'égalité est vérifié pour x = 0.

- Si $|x| \leq 1$, alors $1+x \leq e^x \leq 1+x+x^2$.

On s'intéresse au comportement asymptotique quand $x \to 0$, on a : $e^x = 1 + x + \Theta(x^2)$

et pour tout x : $\lim_{n \longrightarrow \infty}(1 + \frac{x}{n})^n = e^x$

Chapitre 3: Outils mathématiques

- ***Fonctions classiques***
  - ***La fonction logarithme***

Le logarithme d'un nombre dans une base c'est l'exposant qu'il faut donner à la base pour obtenir ce nombre. (exemple $2^3 = 8 \Rightarrow \log_2 8 = 3$).

Notation :

Logarithme népérien : $\log_e n = \ln n$

Logarithme binaire : $\log_2 n = \ln n / \ln 2$

Exponentiation : $\log^K n = (\log n)^K$

Composition : log log n = log (log n).

Chapitre 3: Outils mathématiques

- ***Fonctions classiques***
  - ***La fonction logarithme***

Propriétés de calcul: Pour tout réels a, b, c >0, et un entier n :

1. $a = b^{\log_b a}$
2. $\log_c(ab) = \log_c a + \log_c b$
3. $\log_b a^n = n \log_b a$
4. $\log_b a = \frac{\log_c a}{\log_c b}$
5. $\log_b(1/a) = -\log_b a$
6. $\log_b a = \frac{1}{\log_a b}$
7. $a^{\log_b n} = n^{\log_b a}$

Ainsi le changement de base ne modifie pas la valeur du logarithme que d'un facteur constant, qui est négligeable dans la notation O.

- ***<u>Méthodes de résolutions des équations de récurrence:</u>***

Généralement le temps d'exécution d'une fonction récursive s'exprime par une équation de récurrence. L'équation de récurrence décrit une fonction à partir de sa valeur sur des entrées plus petites.

- ***<u>Méthodes de résolutions des équations de récurrence:</u>***
  - ***<u>Méthode par substitution</u>***
- On pose l'hypothèse d'une borne qu'on devine
- et on démontre par récurrence que cette hypothèse est correcte.
- La difficulté de cette méthode tient au fait qu'il n'existe pas de méthode générale pour deviner une borne autre que l'expérience des cas similaires.

- ***<u>Méthodes de résolutions des équations de récurrence:</u>***
  - ***<u>Méthode par substitution(suite)</u>***

<u>Exemple:</u> Démontrons l'existence d'une borne pour l'expression suivante: $T(n) = 2T(\lfloor n/2 \rfloor) + n$

*On suppose que la solution c'est O(n long n). La méthode consiste à prouver que T(n) ≤ c * n log n pour c > 0.*

On suppose que cette borne est valable pour $\lfloor n/2 \rfloor$.

Donc T($\lfloor n/2 \rfloor$*) ≤ c ** $\lfloor n/2 \rfloor$ *log* $\lfloor n/2 \rfloor$. *On remplace T(*$\lfloor n/2 \rfloor$*) par sa valeur dans l'équation de récurrence et on obtient:*

$$\begin{aligned} T(n) &\leq 2(c * \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq c * n * log(n/2) + n \\ &= c * n \log n - c * n \log 2 + n \\ &= c * n \log n - c * n + n \\ &\leq c * n \log n \quad si\ c \geq 1 \end{aligned}$$

- ***<u>Méthodes de résolutions des équations de récurrence:</u>***
  - ***<u>Méthode par substitution</u>***

<u>Exemple: (suite)</u>

On peut imposer que le cas de base de la récurrence c'est $n = 2$ ou $n = 3$, car pour $n > 3$ la récurrence ne dépend pas de T(Mais cette valeur de c ne convient pas au cas de base de la récurrence car $T(n) \leq c * n \log n = 0$). Il suffit de choisir $c = 2$.

Chapitre 3: Outils mathématiques

- ***Méthodes de résolutions des équations de récurrence:***
  - ***Méthode itérative***

Cette méthode transforme la récurrence en une sommation qu'il s'agit de borner

  - ***Méthode générale***

Elle donne des bornes pour les récurrences de la forme suivante : $T(n) = a*T(n/b)+ f(n)$ avec $a \geq 1$, $b > 1$

Et $f(n)$ est une fonction donnée.

Chapitre 3: Outils mathématiques

Synthèse :

- Rappel des notions mathématiques nécessaire pour le calcul de la complexité.
- Puisque l'objectif de la complexité c'est de définir une borne supérieur au comportement de l'algorithme dans le pire cas, cela revient dans la majeur partie des cas de borner les sommations en se basant sur les approches proposés dans ce chapitre.
- Les approches proposés ne sont qu'une méthode parmi tant d'autres pour définir les bornes d'une sommation.
- L'utilisation de telle ou telle approche ne peut se faire que si les conditions de son applicabilité son vérifier
- C'est l'expérience des cas similaire qui permet de bien choisir l'approche à suivre pour trouver une telle borne.

Chapitre 4: Analyse d'algorithme de tri

- Tri par bulle
- Tri par insertion
- Tri par sélection
- Le tri rapide

Chapitre 4: Analyse d'algorithme de tri

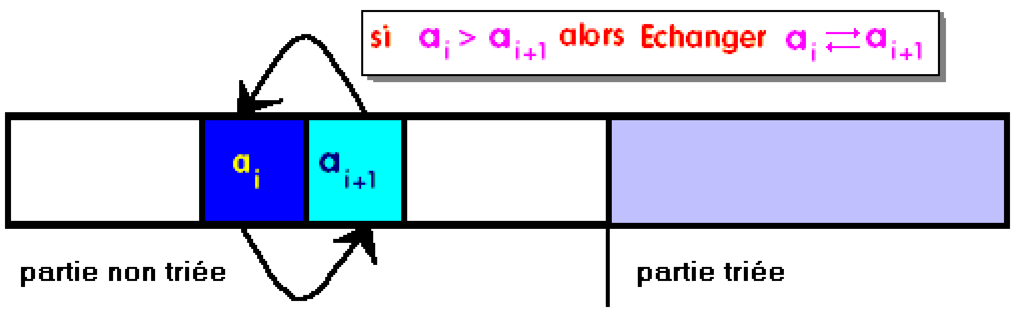
Tri par bulle:

- C'est le moins performant de la catégorie des tris par échange ou sélection, mais comme c'est un algorithme simple, il est intéressant pour une utilisation pédagogiquement.
- Son principe est de parcourir la liste (**a**1, **a**2, ... , **a**n) en intervertissant toute paire d'éléments consécutifs (**a**i-1, **a**i) non ordonnés. Ainsi après le premier parcours, l'élément maximum se retrouve en $\mathbf{a}_n$. On suppose que l'ordre s'écrit de gauche à droite (à gauche le plus petit élément, à droite le plus grand élément).
- On recommence l'opération avec la nouvelle sous-suite (**a**1, **a**2, ... , **a**n-1) , et ainsi de suite jusqu'à épuisement de toutes les sous-suites (la dernière est un couple).
- Le nom de tri à bulle vient du fait qu'à la fin de chaque itération interne, les plus grands nombres de chaque sous-suite se déplacent vers la droite successivement comme des bulles de la gauche vers la droite.

Chapitre 4: Analyse d'algorithme de tri

Tri par bulle: Concrètement

- La suite (**a**1, **a**2, ... , **a**n) est rangée dans un tableau T[...] en mémoire centrale. Le tableau contient une partie triée (en violet à droite) et une partie non triée (en blanc à gauche). On effectue plusieurs fois le parcours du tableau à trier; le principe de base étant de réordonner les couples (**ai**-1, **ai**) non classés (en inversion de rang soit **ai**-1 > **ai**) dans la partie non triée du tableau, puis à déplacer la frontière (le maximum de la sous-suite (**a**1, **a**2, ... , **a**n-1)) d'une position :



- Tant que la partie non triée n'est pas vide, on permute les couples **non ordonnés** ((**ai**-1, **ai**) tels que **ai**-1 > **ai**)) pour obtenir le maximum de celle-ci à l'élément frontière. C.-à-d., qu'au premier passage c'est l'extremum global qui est bien classé, au second passage le second extremum etc...

Tri par bulle: Algorithme

```
Algorithme Tri_a_Bulles
 local:   i , j , n, temp ∈  Entiers naturels
 Entrée : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
 Sortie : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
début
 pour i de n jusquà 1 faire // recommence une sous-suite (a1, a2, ... , ai)
  pour j de 2 jusquà i faire // échange des couples non classés de la sous-suite
   si Tab[ j-1 ] > Tab[ j ] alors // aj-1et aj non ordonnés
     temp ← Tab[ j-1 ] ;
     Tab[ j-1 ] ← Tab[ j ] ;
     Tab[ j ]  ← temp //on échange les positions de aj-1et aj
   Fsi
  fpour
 fpour
Fin Tri_a_Bulles
```

Pire cas : tableau classé dans l'ordre inverse

Complexité = $O(n^2)$.

Chapitre 4: Analyse d'algorithme de tri

i = 6 / pour j de 2 jusqu'à 6 faire

5	4	2	3	7	1	5 > 4 donc permutation des deux cellules
4	5	2	3	7	1	5 > 2 donc permutation des deux cellules
4	2	5	3	7	1	5 > 3 donc permutation des deux cellules
4	2	3	5	7	1	5 < 7 donc aucune action sur ces deux cellules
4	2	3	5	7	1	7 > 1 donc permutation des deux cellules
4	2	3	5	1	7	A la fin de la boucle externe le max 7 est rangé

i = 5 / pour j de 2 jusqu'à 5 faire

4	2	3	5	1	7	4 > 2 donc permutation des deux cellules
2	4	3	5	1	7	4 > 3 donc permutation des deux cellules
2	3	4	5	1	7	4 < 5 donc aucune action sur ces deux cellules
2	3	4	5	1	7	5 > 1 donc permutation des deux cellules
2	3	4	1	5	7	A la fin de la boucle externe le max 5 est rangé

i = 4 / pour j de 2 jusqu'à 4 faire

2	3	4	1	5	7	2 < 3 donc aucune action sur ces deux cellules
2	3	4	1	5	7	3 < 4 donc aucune action sur ces deux cellules
2	3	4	1	5	7	4 > 1 donc permutation des deux cellules
2	3	1	4	5	7	A la fin de la boucle externe le max 4 est rangé

i = 3 / pour j de 2 jusqu'à 3 faire

2	3	1	4	5	7	2 < 3 donc aucune action sur ces deux cellules
2	3	1	4	5	7	3 > 1 donc permutation des deux cellules
2	1	3	4	5	7	A la fin de la boucle externe le max 3 est rangé

i = 2 / pour j de 2 jusqu'à 2 faire

2	1	3	4	5	7	2 > 1 donc permutation des deux cellules
1	2	3	4	5	7	A la fin de la boucle externe le max 2 est rangé

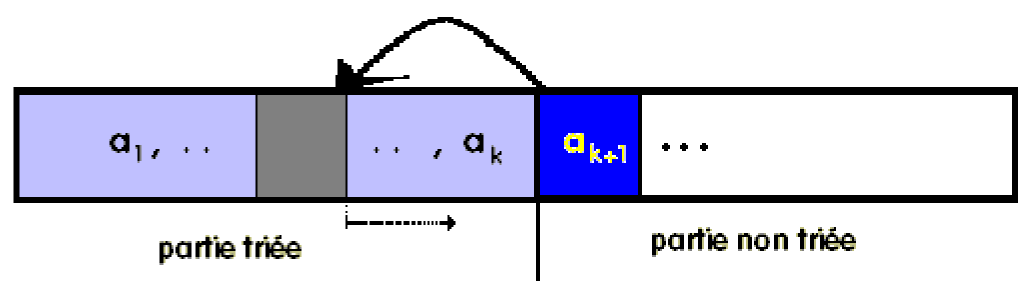
i = 1 / pour j de 2 jusqu'à 1 faire *(boucle vide)*

1	2	3	4	5	7	Il ne reste plus d'éléments à comparer !

Chapitre 4: Analyse d'algorithme de tri

<u>Tri par insertion:</u>

- En général un peu plus coûteux qu'un tri par sélection
- Son principe est de parcourir la liste non triée (**a**1, **a**2, ... , **a**n) en la décomposant en deux parties une partie déjà triée et une partie non triée.
- Identique au rangement des cartes : on insère dans le paquet de cartes déjà rangées une nouvelle carte au bon endroit. L'opération de base consiste à prendre l'élément frontière dans la partie non triée, puis à l'insérer à sa place dans la partie triée (place que l'on recherchera séquentiellement), puis à déplacer la frontière d'une position vers la droite. Ces insertions s'effectuent tant qu'il reste un éléments à ranger dans la partie non triée.. L'insertion de l'élément en frontière est effectuée par décalages successifs d'une cellule.



Chapitre 4: Analyse d’algorithme de tri

Tri par insertion: Algorithme

```
Algorithme Tri_Insertion
 local  i , j , n, v ∈  Entiers naturels
 Entrée : Tab ∈  Tableau d'Entiers naturels de 0 à n éléments
 Sortie : Tab ∈  Tableau d'Entiers naturels de 0 à n éléments
{ dans la cellule de rang 0 se trouve une sentinelle chargée d'éviter de tester dans la boucle
tantque .. faire si l'indice j n'est pas inférieur à 1, elle aura une
valeur inferieure à toute valeur possible de la liste
}
début
 pour i de 2 jusqu à n faire// la partie non encore triée (ai, ai−1, ... , an)
    v ← Tab[ i ] ;  // l'élément frontière : ai
    j ← i ;         // le rang de l'élément frontière
     Tantque Tab[ j-1 ] > v faire//on travaille sur la partie déjà triée (a1, a2, ... , ai)
           Tab[ j ]  ← Tab[ j-1 ]; // on décale l'élément
            j ← j-1;             // on passe au rang précédent
     FinTant ;
     Tab[ j ]  ← v //on recopie ai dans la place libérée
  fpour
Fin Tri_Insertion
```

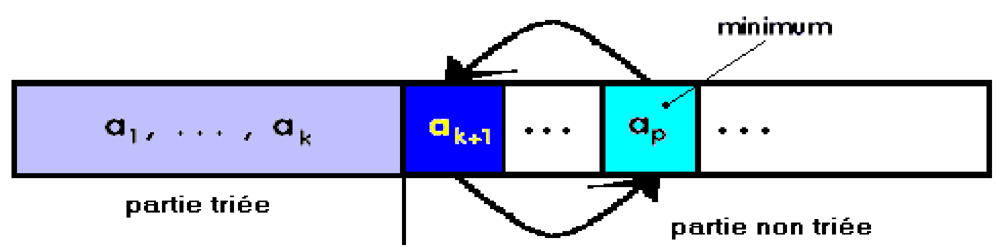
Pire cas : Tableau ordonnée dans le sens inverse.

Nombre d’opération = $n(n+1)/2 + 2(n-1) \rightarrow O(N) = N^2$

Tri par sélection:

- Le principe est de parcourir la partie non-triée de la liste (**a**k+1, **a**k+2, ... , **a**n) en cherchant l'élément minimum, puis en l'échangeant avec l'élément **a**k+1, puis à déplacer la frontière d'une position. Il s'agit d'une récurrence sur les minima successifs. On suppose que l'ordre s'écrit de gauche à droite.
- On recommence l'opération avec la nouvelle sous-suite (**a**k+2, ... , **a**n) , et ainsi de suite jusqu'à ce la dernière soit vide.

si $a_{k+1} > a_p$ alors Echanger $a_p \rightleftarrows a_{k+1}$



Chapitre 4: Analyse d'algorithme de tri

Tri par sélection: Algorithme

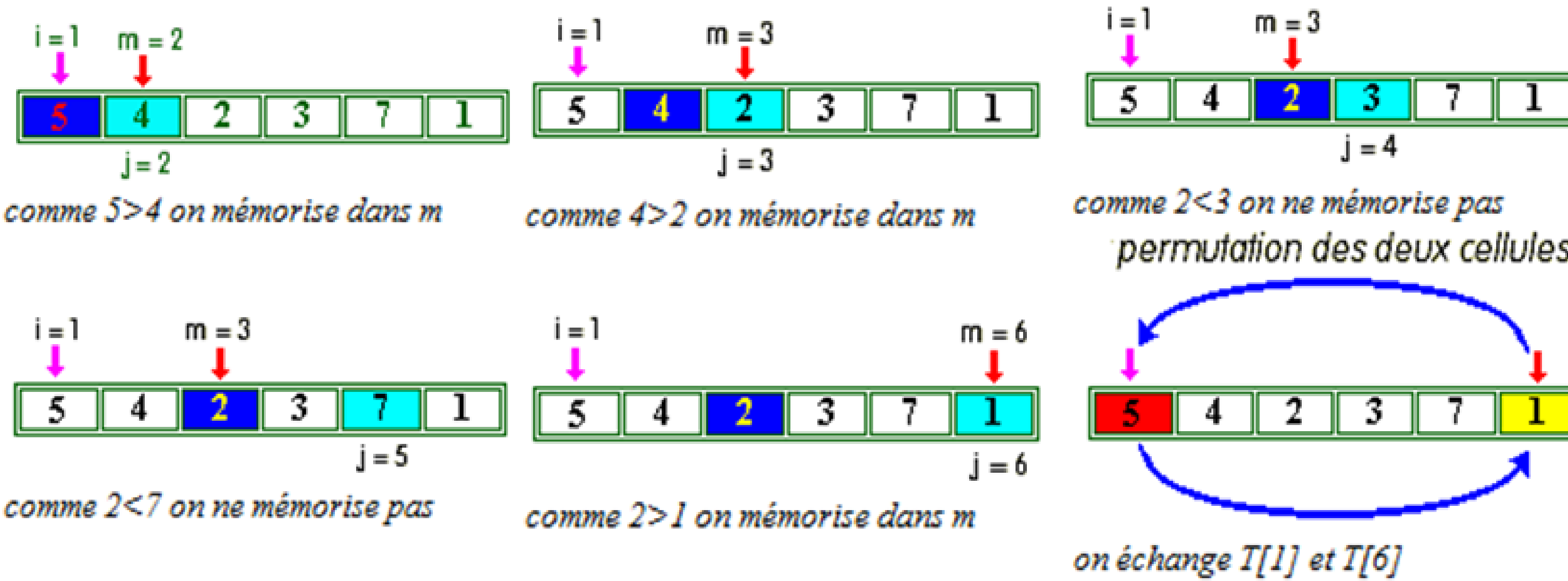
```
Algorithme Tri_Selection
 local:   m, i , j , n, temp ∈  Entiers naturels
 Entrée : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
 Sortie : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments

début
 pour i de 1 jusquà n-1faire // recommence une sous-suite
  m ← i ; // i est l'indice de l'élément frontière ai = Tab[ i ]
  pour j de i+1 jusquà n faire// (ai+1, a2, ... , an)
   si Tab[ j ] <  Tab[ m ] alors // aj est le nouveau minimum partiel
    m ← j ; // indice mémorisé
   Fsi
  fpour;
  temp ← Tab[ m ] ;
  Tab[ m ] ← Tab[ i ] ;
  Tab[ i ]  ← temp //on échange les positions de ai et de aj
 fpour
Fin Tri_Selection
```

Complexité = $O(N^2)$

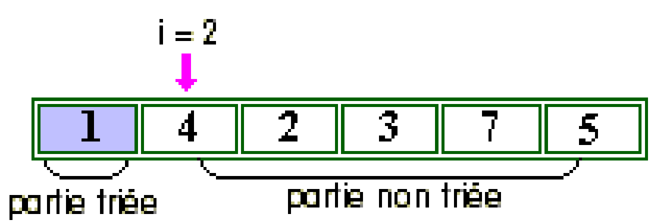
Chapitre 4: Analyse d'algorithme de tri

Tri par sélection: Exemple



L'algorithme ayant terminé l'échange de T[1] et de T[6], il passe à l'itération externe suivante (i = 2) :

etc….



Tri rapide:

- C'est le plus performant des tris en table et certainement le plus utilisé. Ce tri a été trouvé par C.A.Hoare.
- Son principe est de parcourir la liste L = (**a**1, **a**2, ... , **a**n) en la divisant systématiquement en deux sous-listes L1 et L2. L'une est telle que tous ses éléments sont inférieurs à tous ceux de l'autre liste et en travaillant séparément sur chacune des deux sous-listes en réappliquant la même division à chacune des deux sous-listes jusqu'à obtenir uniquement des sous-listes à un seul élément.
- C'est un algorithme dichotomique qui divise donc le problème en deux sous-problèmes dont les résultats sont réutilisés par recombinaison, il est donc de complexité **O(n.log(n))**.

<u>Tri rapide:</u> Principe 1/3

- Pour partitionner une liste L en deux sous-listes L_1 et L_2 :
  - on choisit une valeur quelconque dans la liste L appelée pivot,
  - La sous-liste L1 va contenir tous les éléments de L inférieure ou égale au pivot,
  - et la sous-liste L2 tous les éléments de L supérieure au pivot.
- Ainsi de proche en proche en subdivisant le problème en deux sous-problèmes, à chaque étape, nous obtenons un pivot bien placé.
- Le processus de partitionnement décrit ci-haut (appelé aussi segmentation) est le point central du tri rapide.
- Une fonction **Partition** va réaliser cette action .Cette fonction est récursive puisqu'on la réapplique sur les deux sous-listes obtenues, le tri rapide devient alors une procédure récursive.

Tri rapide: Principe 2/3

L = [4, 23, 3, 42, 2, 14, 45, 18, 38, 16]

prenons comme pivot la dernière valeur pivot = 16

- Nous obtenons par exemple :
  L1 = [4, 3, 2, 14]
  L2 = [23, 45, 18, 38, 42]
- **A cette étape voici l'arrangement de L :**
  L = L1 + pivot + L2 = [**4,, 3, 2,14,16**, **23, 45, 18, 38, 42**]
- En effet, en travaillant sur la table elle-même par réarrangement des valeurs, le pivot **16** est placé au bon endroit directement :
  [**4<16, 3<16, 2<16**, **14<16,16**, **23>16, 45>16, 18>16, 38>16, 42>16**]

Chapitre 4: Analyse d'algorithme de tri

Tri rapide: Principe 3/3

- En appliquant la même démarche au deux sous-listes : L1 (pivot=2) et L2 (pivot=42)
  [**4, 14, 3, 2**, **16**, **23, 45, 18, 38, 42**] nous obtenons :
- L11=[ ] liste vide
  L12=[**4, 3, 14**]
  L1=L11 + pivot + L12 = (**2,4, 3, 14**)
- L21=[**23, 38, 18**]
  L22=[**45**]
  L2=L21 + pivot + L22 = (**23, 38, 18, 42, 45**)
- **A cette étape voici le nouvel arrangement de L :**
  L = [(**2,4, 3, 14**), **16**, (**23, 38, 18, 42, 45**)]
  etc…

Chapitre 4: Analyse d'algorithme de tri

Tri rapide

```
Global :Tab[min..max] tableau d'entier

fonction Partition( G , D : entier )  résultat : entier
Local : i , j ,  piv , temp : entier
début
    piv  ←  Tab[D];
    i  ← G-1;
    j  ← D;
    repeter
      repeter  i ←  i+1 jusquà Tab[i] >= piv;
      repeter  j ←  j-1 jusquà Tab[j] <= piv;
      temp ←  Tab[i];
      Tab[i] ←  Tab[j];
      Tab[j] ←  temp
    jusquà j <= i;
    Tab[j] ←  Tab[i];
    Tab[i] ←  Tab[d];
    Tab[d] ←  temp;
    résultat ←  i
FinPartition

Algorithme TriRapide( G  , D : entier );
Local :  i : entier
début
 si D > G alors
   i ← Partition( G , D );
  TriRapide( G , i-1 );
  TriRapide( i+1  , D );
 Fsi
FinTRiRapide
```

Chapitre 5: Structures de données avancées

- Pile et File
- Liste
- Graphe
- Arbre
- Dictionnaire
- Index
- Hachage et table de Hachage

Chapitre 5: Structures de données avancées

- ***Pile***
  - Une structure de données mettant en œuvre le principe LIFO : Last In First Out
  - Une pile P peut être implémentée par un tableau, et elle est caractérisée par :
    - Un sommet noté sommet(P) indiquant l'indice de l'élément le plus récemment inséré dans la pile
    - Un caractère spécial, comme $, initialisant la pile
    - Une procédure EMPILER(P,x)
    - Une fonction DEPILER(P)
    - Une fonction booléenne PILE-VIDE(P) retournant VRAI si et seulement si P est vide

Chapitre 5: Structures de données avancées

- ***Pile***

fonction PILE-VIDE(P: pile): bouléen
Début Si sommet(P)=$ alors retourner VRAI
        sinon retourner FAUX Fsi;
Fin.

Procédure EMPILER(P,x)
Début Si sommet(P)=longueur(P)
        alors erreur (débordement positif)
        sinon sommet(P)=sommet(P)+1; P[sommet(P)]=x; fsi;
Fin.

Fonction DEPILER(P): élément
Début Si PILE-VIDE(P)
        alors erreur (débordement négatif)
        sinon sommet(P)=sommet(P)-1; x = P[sommet(P)+1] ; fsi;
Fin.

- ***File***
  - Une structure de données mettant en œuvre le principe FIFO : First In First Out.
  - Une file F peut être implémentée par un tableau, et elle est caractérisée par :
    - Un pointeur tête(F) qui pointe vers la tête de la file (l'élément le plus anciennement inséré)
    - Un pointeur queue(F) qui pointe vers la première place libre, où se fera la prochaine insertion éventuelle d'un élément
    - Initialement : tête(F)=NIL et queue(F)=1

- ***File***
- fonction FILE-VIDE(P: pile): bouléen

Début Si tete(F)=NIL alors retourner VRAI
    sinon retourner FAUX Fsi;
Fin.

- Procédure INSERTION(P,x)

Début si [tête(F)≠NIL et queue(F)=tête(F)]
    alors erreur (débordement positif)
    sinon F[queue(F)]=x; queue(F)=[queue(F)+1](modulo n);
        si[tête(F)=NIL] alors tête(F)=1fsi; fsi;
Fin.

- Fonction DEPILER(P): élément

Début si FILE-VIDE(F) alors erreur (débordement négatif)
    sinon temp=F(tête(F)); tête(F)=[tête(F)+1](modulo n);
        si[tête(F)=queue(F)] alors tête(F)=NIL ; queue(F)=1;
    fsi; fsi;
retourner temp;
Fin.

- ***<u>Liste chainée</u>***
- Une liste chaînée est une structure de données dont les éléments sont arrangés linéairement, l’ordre linéaire étant donné par des pointeurs sur les éléments
- Un élément d’une liste chaînée est un enregistrement contenant un champ clé, un champ successeur consistant en un pointeur sur l’élément suivant dans la liste
- Si le champ successeur d’un élément vaut NIL, il s’agit du dernier élément de la liste, appelé aussi queue de la liste
- Un pointeur TETE(L) est associé à une liste chaînée L : il pointe sur le premier élément de la liste
- Si une liste chaînée L est telle que TETE(L)=NIL alors la liste est vide

- ***<u>Liste chainée particulière :</u>***
- Liste doublement chainée
- Liste chainée triée
- Liste circulaire (anneau)

- ***Liste chainée :***

Algorithme de manipulation des listes simplement chainées

```
RECHERCHE-LISTE(L,k)
Début x=TETE(L);
tant que(x<>NIL et clé(x)!=k)
        faire x=x -> svt; fait;
retourner x;
Fin;
```

```
INSERTION-LISTE_1(L,X)
Début x->svt =TETE(L);
        TETE(L)=x;
Fin;
```

```
INSERTION-LISTE_2(L,X)
Début
Y=tete(L);
Tant que (y->svt <>NIL)
Faire y=y -> svt; fait;
y->svt=x;
Fin;
```

```
INSERTION-LISTE_3(L,X)
Début
Y=tete(L);
Tant que (y->svt <>NIL)et (y.cle < x.cle)
Faire z:= y; y=y -> svt; fait;
z->svt=x;
x->svt=y;
Fin;
```

- ***<u>Liste chainée :</u>***

<u>Algorithme de manipulation des listes simplement chainées</u>

```
SUPPRESSION-LISTE_1(L,X)
Début
        Si x=TETE(L) alors TETE(L)=x->svt;
        sinon y=TETE(L);
        tant que (y.cle<>x.cle) et (y->svt <> NIL)
        faire x=y; y=y->svt; fait;
        Y->svt=x->svt;Fsi;
Fin;
```

```
SUPPRESSION-LISTE_2(L,k)
Début
Si k<0 ou k> langueur(L) alors erreur
Sinon i=1;y=tete(L); tant que i< k faire faire z=y; y=y->svt; fait;
        z->svt=y->svt;Fsi;
Fin;
```

- ***Liste chainée :***

Algorithme de manipulation des listes simplement chainées

Exercice:

Proposez un algorithme de tri d'une liste

- ***<u>Graphe :</u>***
- **Graphe orienté :** couple (S,A), S ensemble fini (sommets) et A relation binaire sur S (arcs)
- **Graphe non orienté :** couple (S,A), S ensemble fini (sommets) et A relation binaire sur S (arêtes)

- ***Graphe :***

Quelque propriétés sur les graphes

- Arcs : un arc est orienté (couple)
- Arêtes : une arête n'est pas orientée (paire)
- Possibilité d'avoir des boucles (une boucle est un arc reliant un sommet à lui-même) ou pas
- Un arc (u,v) part du sommet u et arrive au sommet v
- Une arête (u,v) est incidente aux sommets u et v
- Degré sortant d'un sommet : nombre d'arcs en partant
- Degré rentrant d'un sommet : nombre d'arcs y arrivant
- Degré= degré sortant + degré entrant
- Degré d'un sommet : nombre d'arêtes qui lui sont incidentes

- ***Graphe :***

Quelque propriétés sur les graphes

- Chemin de degré k d'un sommet u à un sommet v : (u0,u1,…,uk) u=u0 et v=uk
- Chemin élémentaire : plus court chemin
- Chaîne et chaîne élémentaire :suite d'arête reliant deux sommets
- Circuit et circuit élémentaire : chemin fermé
- Cycle et cycle élémentaire : circuit non orienté
- Graphe sans circuit
- Graphe acyclique : graphe ne contenant aucun cycle.

- ***<u>Graphe :</u>***

<u>Quelque propriétés sur les graphes</u>

- Graphe fortement connexe : tout sommet est accessible à partir de tout autre sommet (par un chemin)
- Graphe connexe : chaque paire de sommets est reliée par une chaîne
- Composantes fortement connexes d'un graphe : classes d'équivalence de la relation définie comme suit sur l'ensemble des sommets : R(s1,s2) si et seulement si il existe un chemin (resp. chaîne) de s1 vers s2 et un chemin (resp. chaîne) de s2 vers s1.

- ***Arbre :***
- G=(S,A) graphe non orienté :
- G arbre
- G connexe et |A|=|S|-1
- G acyclique et |A|=|S|-1
- Deux sommets quelconques sont reliés par une unique chaîne élémentaire

Arbre enracinée (rooted tree)

- La racine de l'arbre: un sommet particulier
- La racine impose un sens de parcours de l'arbre
- Pour un arbre enraciné : sommets ou nœuds
- Ancêtre, père, fils, descendant
- L'unique nœud sans père : la racine
- Nœuds sans fils : nœuds externes ou feuilles
- Nœuds avec fils : nœuds internes
- Sous arbre en x : l'arbre composé des descendants de x.
- Degré d'un nœud : nombre de fils
- Profondeur : longueur du chemin entre la racine et le nœud
- Hauteur d'un arbre
- Arbre ordonné

Arbre binaire

- De manière récursive ce défini par :
  - Ne contient aucun nœud (arbre vide)
  - Est formé de trois ensembles disjoints de nœuds : la racine ; le sous arbre gauche (arbre binaire) ; le sous arbre droit (arbre binaire)
- Un arbre binaire est plus qu’un arbre ordonné
- Arbre binaire complet : au moins 0 fils, au plus 2 fils
- Arbre n-aire : degré d’un noeud≤n.
- Un arbre binaire peut être représenté par une liste doublement chainée

Arbre binaire: Parcours

- Par profondeur d'abord
  - Descendre le plus profondément possible dans l'arbre
  - Une fois une feuille atteinte, remonter pour explorer les branches non encore explorées, en commençant par la branche la plus basse
  - Les fils d'un nœud sont parcourus suivant l'ordre défini sur l'arbre
- Par largeur d'abord
  - Visiter tous les nœuds de profondeur i avant de passer à la visite des nœuds de profondeur i+1
  - Utilisation d'une file

Algorithme profondeur d'abord (Préfixé)

- Algorithme Parcours_Préfixé (Arbre Racine)

```
Début
   A:=Racine; Afficher(A.clé);  Empiler(A,P);
   Tant que (non Vide(P))
      A:=A->fg;
      Tant que (A -> <>null) Faire Afficher(A.clé);  Empiler(A,P);
                                    A:=A->fg;
                        Fait;
      A:=Depiler(P); /* cas des feuilles*/
      Si tete(P)-> fg = A alors A := Tete(P)->fd;
                  Sinon tant que ((tete(P)->fd = A) et non vide(P))
                         faire A:=Depiler(P); fait;
                         Fait;
      Fsi;
   Fait;
Fin.
```

Arbre binaire de recherche

- *Définition* : Un arbre binaire de recherche est un arbre binaire tel que pour tout nœud x :
  - Tous les nœuds y du sous arbre gauche de x vérifient clef(y)<clef(x)
  - Tous les nœuds y du sous arbre droit de x vérifient clef(y)>clef(x)
- *Propriété* : le parcours (en profondeur d'abord) infixe d'un arbre binaire de recherche permet l'affichage des clés des nœuds par ordre croissant

```
Infixe(A)
début
        Si A≠NIL alors Infixe(A.sa-gauche);
                       Afficher(A.clef);
                       Infixe(A.sa-droit);
Fsi; FIN;
```

Arbre binaire de recherche

Recherche d'un élément :

- la fonction rechercher(A,k) ci-dessous retourne le pointeur sur un nœud dont la clé est k, si un tel nœud existe
- elle retourne le pointeur NIL sinon

```
rechercher(A,k)
Début
si (A=NIL ou A.clef=k) alors retourner A
sinon si (k<A.clef) alors rechercher(A.sa-gauche,k)
                          sinon rechercher(A.sa-droit,k)
                          Fsi;
Fsi;
Fin;
```

Arbre binaire de recherche

Insertion d'un élément :

- la fonction insertion(A,k) insère un nœud de clé k dans l'arbre dont la racine est pointée par A.

```
Insertion(A,k)
Début
z.clef=k; z.sa-gauche=NIL; z.sa-droit=NIL;
x=A; père_de_x=NIL;
tant que (x≠NIL) faire père_de_x=x;
                        si (k<x.clef) alors x=x.sa-gauche
                                   sinon x=x.sa-droit fsi;
si (père_de_x=NIL) alors A=z
sinon si (k<père_de_x.clef) alors père_de_x.sa-gauche=z
      sinon père_de_x.sa-droit=z Fsi;
Fin.
```

Index et Dictionnaire

- Index et dictionnaire sont des structures de données pour représenter un ensemble de mots et rechercher l'appartenance ou non d'un mot à l'ensemble.

- La différence fondamentale entre ces deux structures c'est la données conservée, la taille de la structure et la complexité des algorithmes pour les opérations de manipulations.

Dictionnaire

- Un dictionnaire est un ensemble de mots qui supporte uniquement les opérations suivantes : Rechercher un mot, Insérer un mot, Supprimer un mot
- L'opération de recherche est considérée comme la plus importante. C'est l'unique opération si le dictionnaire est statique, s'il est dynamique les opérations de consultations sont généralement très supérieurs au nombre d'ajouts et de suppression de mots, d'où son importance.
- A tout ensemble de mots fini E = $\{m_1, m_2, \ldots, m_n\}$, on associe une structure Dictionnaire(E) qui, pour un mot M donné, vérifie son appartenance au dictionnaire, retourne sa position s'il existe et (-1) ou (0) sinon.

Index

- Un index est une structure permettant de représenter les occurrences (positions) d'un ensemble de mot d'un texte ou d'un ensemble de textes.
- Rechercher un mot dans le texte consiste à retrouver, soit sa première occurrence, soit toutes les occurrences, soit une occurrence à partir d'une position donnée du texte, soit dans un intervalle de position.
- L'index est dynamique si on peut ajouter de nouveaux mots à l'index actuel.
- Les opérations de base dans un dictionnaire sont généralisés dans un index.
- Elles se basent aussi sur les techniques de hachage pour une implémentation plus efficace

Index

- Soit $E = \{T_1, T_2, \ldots, T_n\}$ un ensemble de n textes. On définit l'index de E par l'ensemble suivant :
  - Index (E) = {(m, p, i) | m est un mot ou un facteur dans un moins un texte (T_i, i = 1, … , n) et p une position de m dans un $T_i \in E$
- Donc si $E = \{T_1, T_2, \ldots, T_n\}$ est un ensemble de n textes, l'index de ces n textes est un ensemble de mots, tel que chaque mot a une occurrence dans au moins un texte et p est la position de cette occurrence.

Index

- Opération sur les index
  - ***La recherche:*** trouver toutes les positions d'un mot dans un texte
  - ***La mise à jour:*** Ajouter ou supprimer un texte à l'ensemble des textes
    - Dans un index statique cela revient à réindexer (reconstruire un nouvel index) après la mise à jour
    - Dans un index dynamique (incrémental)

Le hachage

- Une table de hachage est une structure qui permet d'implémenter un dictionnaire.
  - Soit T une table de hachage (dictionnaire) et x un mot de T. Une table de hachage permet d'associer une clé d'accès *f(x)* à chaque élément x de T
- La recherche d'une clé dans T est au O(n), n étant le nombre d'entrées dans la table T, mais en pratique on peut arriver à des algorithmes en O(1) pour un élément qui existe dans la table
- La table de hachage générale la notion de tableau classique. L'accès direct à un élément i du tableau se fait en O(1). Cette généralisation est rendu possible grâce à la notion de clé

Le hachage

- La clé ou la position d'un élément dans la table T, se calcule à l'aide d'une fonction f, dite fonction de hachage.
- Cette fonction prend en entrée un élément x de T et fournit en sortie un entier dans un intervalle donné.
- La fonction f est construite de manière à ce que les entiers qu'elle fournit soient uniformément distribués dans cet intervalle. Ces entiers sont les indices du tableau T.
- Les tables de hachage sont très utiles lorsque le nombre de données effectivement stockées et très petit par rapport au nombre de données possibles

Le hachage

- ***Exemple :*** Soit T le tableau suivant :

7	5	4	6	1	8	2	3

- La recherche dans ce tableau est en O(n). Mais si on constate que tous les T[i] <= 8, on peut obtenir une recherche en O(1) avec le tableau suivant :

1	2	3	4	5	6	7	8

- Mais si n est grand et le nombre de clés présentés dans T est m avec m << n, on ne peut plus adopter cette solution, on utilise les tables de hachage.

C'est le cas d'un dictionnaire de la langue. L'alphabet est $\sum = \{a, b, c, d, \ldots, y, z\}$, $|\sum| = 26$, le nombre de mots possibles obtenus par combinaisons et permutations des caractères est très grand par rapport au nombre réel des mots d'un dictionnaire. L'arrangement de r lettres parmi 26, donne A^r_{26} mots possible, . Si r tend vers 26 (le mot le plus long), ce nombre tend vers 26! mots possibles.

Le hachage

- On dit qu'il y'a collision si deux éléments différents x_1 et x_2 de T peuvent avoir la même clé , $f(x_1) = f(x_2)$
  - Exemple : Fonction de hachage = mod(n).
- Une bonne fonction de hachage doit minimiser le nombre de collisions
- Un algorithme complet de hachage consiste en une fonction de hachage et une méthode de résolution des collisions
- Il existe deux grandes classes distinctes de méthodes de résolution de collisions

Le hachage

- ***Méthode de résolution des collisions :***
  - Hachage par chaînage : En cas de collisions, tous les éléments accessibles par la même clés seront chaînés entre eux pour faciliter d'accès et les opérations de mise à jour.
  - Adressage ouvert : Dans cette méthode, tous les éléments seront conservés dans la table de hachage elle-même. Chaque entrée de la table contient soit une clé, soit NUL. Si k est une clé, on vérifie si f(k) est dans la table ou non, si on ne le trouve pas on calcule un nouvel indice à partir de cette clé (re-hache) jusqu'à trouver la clé ou NUL si l'élément n'appartient pas à la table. Dans ce type de hachage, la table peut se remplir entièrement et on ne peut plus insérer une nouvelle clé

Le hachage

- Type de hachage :
  - Statique : table de taille fixe
  - Dynamique : extension du hachage statique pour s'adapter à des tables de tables croissantes ou décroissantes
  - Parfait : taux de collision = 0
  - Table de hachage distribué (DHT): identifier et retrouver une information dans un système réparti, comme les réseaux P2P.

Synthèse:

Chapitre 6: Stratégie de résolution des problèmes

1. ***Approche par force brute:***
   - Résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
2. ***Diviser pour régner:***
   - Diviser le problème à résoudre en un certain nombre de sous-problèmes (plus petits)
   - Régner sur les sous-problèmes en les résolvant récursivement :
     - Si un sous-problème est élémentaire (indécomposable), le résoudre directement
     - Sinon, le diviser à son tour en sous-problèmes
     - Combiner les solutions des sous-problèmes pour construire une solution du problème initial

3. ***<u>Programmation dynamique:</u>***
   - Obtenir la solution optimale à un problème en combinant des solutions optimales à des sous-problèmes similaires plus petits et se chevauchant
4. ***<u>Algorithmes gloutons:</u>***
   - Construire la solution incrémentalement, en optimisant de manière aveugle un critère local

Chapitre 7: Les classes de problèmes

Type de problèmes

Un problème est étroitement lié à la question posée dans sa définition. Selon la forme de la question posée, il existe différents types de problèmes selon le degré de leur difficulté :

- Problème de décision : Réponse 'oui' ou 'non'
- Problème de recherche de solution : Trouver une ou plusieurs solution possible
- Problème d'optimisation : calcul de la solution approchée
- Problèmes de dénombrement de solution : Déterminer le nombre de solutions du problème sans toutefois les rechercher

Les classes de problèmes

Définition 1:

Un algorithme est dit en temps polynômial, si pour tout n, n étant la taille des données, l'algorithme s'exécute en temps borné par un polynôme $f(n) = c*n^k$ opérations élémentaires.

Définition 2:

On dit qu'un problème est polynômial si est seulement si il existe un algorithme polynomiale pour le résoudre.

Définition 3:

Un algorithme est dit ***de résolution*** s'il permet de construire la solution au problème, et il est dit ***de validation*** s'il permet de vérifier si une solution donnée répond au problème.

Les classes de problèmes

Définition 4: Réduction polynomiale:

Un problème P1 peut être ramené ou réduit à un problème P2 si une instance quelconque de P1 peut être traduite comme une instance de P2 et la solution de P2 sera aussi solution de P1. la fonction de traduction ou de transformation doit être polynomiale (De complexité polynomiale).

La relation de réductibilité est réflexive et transitive

La réduction polynomiale de P1 en P2 est noté :

$$P1 \leq P2$$

Les classes de problèmes

Quelle est la particularité d'un algorithme polynômial par rapport à un algorithme exponentiel ?

Chapitre 7: Les classes de problèmes

Les classes de problèmes

La classe P

Un problème est de classe P s'il existe un algorithme ***polynomiale déterministe*** pour le résoudre.

La classe NP

Un algorithme est de classe NP si et seulement si, il existe un algorithme de validation ***non déterministe*** et qui s'exécute en un temps ***polynomiale***. Ainsi, la classe NP contient l'ensemble des problèmes dont la vérification est polynomial mais dont la résolution ne l'est pas obligatoirement.

Les problèmes NP-Complet

On dit qu'un problème A est NP-Complet si et seulement si:

1. A appartient à la classe NP
2. $\forall B \in NP, B \leq A$.

Les classes de problèmes

Les problèmes NP-Complet

On dit qu'un problème A est NP-Complet si et seulement si:

1. A appartient à la classe NP
2. $\exists B \in NP - Complet, B \leq A$.

Explication : B est NP-Complet $\Rightarrow \forall X \in NP, X \leq B$

Donc si on arrive à trouver un problème déjà démontrer NP-Complet et qu'on puisse le réduire en notre problème A, on aura démontrer par transitivité que tous les problèmes NP peuvent être réduit en le problème A. Donc, si :

$$\exists B \in NP - Complet, B \leq A \Rightarrow \forall X \in NP, X \leq B \leq A$$

Chapitre 7: Les classes de problèmes

- Quelques problèmes NP-complets
  - SAT : le père des problèmes NP-complets (le tout premier à avoir été montré NP-complet par Stephen COOK en 1971)
  - 3-SAT : Satisfiabilité d'une conjonction de clauses dont chaque clause est composée d'exactement trois littéraux
  - CYCLE HAMILTONIEN : Existence dans un graphe d'un cycle hamiltonien
  - VOYAGEUR DE COMMERCE : Existence dans un graphe pondéré d'un cycle hamiltonien de coût minimal
  - CLIQUE : Existence dans un graphe d'une clique (sous-graphe complet) de taille k
  - 3-COLORIAGE D'UN GRAPHE : peut-on colorier les sommets d'un graphe avec trois couleurs de telle sorte que deux sommets adjacents n'aient pas la même couleur ?
  - PARTITION : Peut-on partitionner un ensemble d'entiers en deux sous-ensembles de même somme ?