Nantes Université

BUT Informatique
BUT2 - R3.02
2024/2025
Christian Attiogbé

Conception d'algorithmes corrects et efficaces (R3.02)

Cahier d'exercices - TD & TP

Novembre 2024

Ressource R3.02 : Développement efficace (cadrage national)

Compétences ciblées :

- Développer — c'est-à-dire concevoir, coder, tester et intégrer — une solution informatique pour un client.
- Proposer des applications informatiques optimisées en fonction de critères spécifiques : temps d'exécution, précision, consommation de ressources.. Descriptif : L'objectif de cette ressource est de renforcer l'apprentissage de l'algorithmique afin d'amener vers une efficacité de développement.

Savoirs de référence étudiés :

- Développement de structures de données complexes (par ex. : collections, arbres, dictionnaires...)
- Premières approches de l'analyse de la performance (profiling, optimisation, greencode...)

Prolongements suggérés :

- Notions de programmation fonctionnelle intégrée à des langages non fonctionnels (lambda-expressions, clôtures...)
- Appréhension des conséquences d'une faille dans le code

Apprentissages critiques ciblés :

- AC21.03 : Adopter de bonnes pratiques de conception et de programmation
- AC22.01 : Choisir des structures de données complexes adaptées au problème
- AC22.02 : Utiliser des techniques algorithmiques adaptées pour des problèmes complexes (par ex. recherche opérationnelle, méthodes arborescentes, optimisation globale, intelligence artificielle...)
- AC22.04 : Évaluer l'impact environnemental et sociétal des solutions proposées

Mots clés : Structure de données – Performance

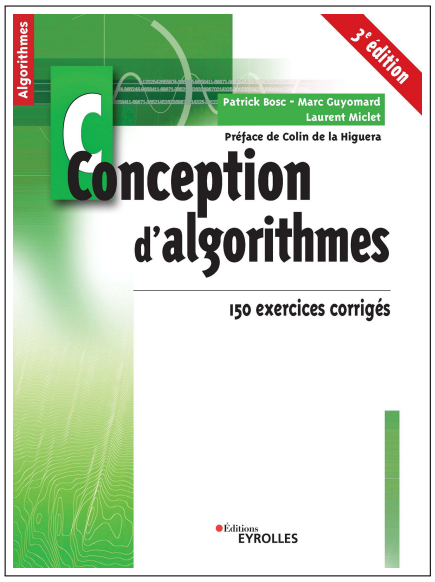
Les exercices proposés dans la suite visent à couvrir les apprentissages critiques ;

AC21.03	✓	AC22.01	✓	AC22.02	✓	AC22.04	✓

Contents

Principale référence utilisée pour cet enseignement :

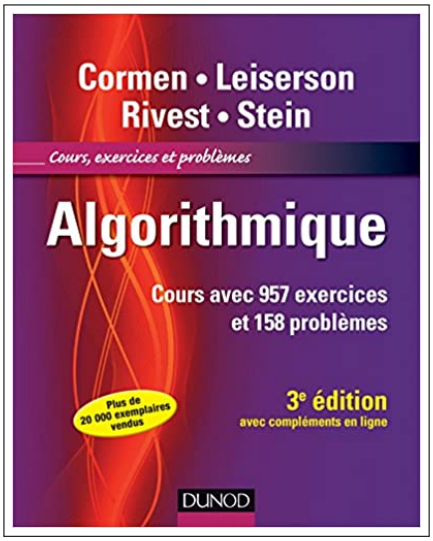
- *Conception d'algorithmes*, Patrick Bosc, Marc Guyomard, Laurent Miclet, Eyrolles, 2021 (**Nous avons tiré la plupart des textes des exercices dans ce livre**).



Déclaration : *Dans ce contexte pédagogique, nous nous sommes permis de reprendre le texte des exercices présentés dans l'ouvrage de référence de l'enseignement. Nous en savons gré aux auteurs (Bosc, Guyomard, Miclet).*

Et, pour prolonger le plaisir :

- *Introduction à l'algorithmique*, T. Cormen, C. Leiserson, R. Rivest, C. Stein, Dunod, 2010
- *Cours et exercices corrigés d'algorithmique*, Jacques Julliand, Vuibert, 2010



1 Introduction et rappels

Les algorithmes comme les logiciels qu'ils engendrent ont des coûts calculables, mesurables ou estimables. Ces coûts permettent également de comparer les algorithmes/logiciels en terme de performance par exemple.

Généralement on considère le coût en temps, le coût en mémoire (et leur impact sur l'environnement : énergie consommée par les processeurs, le réseau), mais on peut quantifier d'autres coûts selon les circonstances et les projets.

En fonction des critères choisis par le programmeur, on peut calculer ou estimer le coût d'un programme/algorithme.

Par exemple, on peut compter :

- les opérations élémentaires : affectation de valeur à une variable, comparaison d'éléments, ...
- les opérations non élémentaires : les accès à la mémoire, les entrées/sorties vers des fichiers ou périphériques, les accès réseau, etc

A partir des temps consommés par les opérations, on estime/calcule le coût en temps de l'algorithme/programme en fonction de la quantité de ces opérations effectuées.

De la même façon, à partir des structures de données utilisées, on estime/calcule l'espace mémoire occupé ou utilisé par un algorithme/programme. Par exemple, les structures de données statiques (Tableaux) et dynamiques (listes, arbres, graphes) n'engendrent pas les mêmes coûts, et ne satisfont pas les mêmes besoins pour les programmeurs.

Pour estimer et analyser les coûts, on utilise les **ordres de grandeur** (cf. cours).

Ordres de grandeur

On compare les algorithmes (et aussi les programmes dérivés), en calculant la quantité de ressources temps ou espaces mémoires utilisées. Ces quantités considérées sont les ordres de grandeur (au lieu des valeurs exactes). On les estime en prenant en compte par exemple les cas extrêmes (favorable, défavorable, pratique).

Formellement le coût d'un algorithme est donné par une fonction $f(n)$.

Avec les ordres de grandeur on majore, ou on encadre $f(n)$ par d'autres fonctions dont les comportements/croissances sont connus. Par exemple, le coût d'un algorithme en fonction de la taille des données n, peut être donné par une fonction $f(n) = n(n+1)/2$

On va donc positionner $f(n)$ par rapport à des fonctions connues au lieu de réétudier complètement $f(n)$

Ordres de grandeur

Un ordre de grandeur est un **ensemble de fonctions** (ou une classe), utilisé comme référence pour apprécier la croissance d'une fonction de coût : n^2, $log_2(n)$, n^p, etc

$$O(g) = \{\, h \in \mathbb{N} \to \mathbb{R}_+ \mid \exists(c, n_0).(c \in \mathbb{R}, n_0 \in \mathbb{R} \ \wedge \\ \forall n.((n \in \mathbb{N} \ \wedge \ n \geq n_0) \ \Rightarrow \ h(n) \leq c.g(n))) \,\}$$

Ainsi, on exprime que le coût $f(n)$ d'un algorithme est dans $O(g(n))$

Quelques ordres de grandeur

Voici quelques ordres de grandeur des mesures de complexité:

Classe	dénomination ou type de complexité
$O(n!)$	complexité factorielle ; complètement démentiel
$O(2^n)$	complexité exponentielle ; ces algorithmes ont un coût catastrophique quand la taille n des données grandit.
$O(n^2)$	complexité quadratique ; de nombreux algorithmes ; acceptables pour des tailles assez grandes des données (> 10-100taines milliers).
$O(nlogn)$	complexité pseudo-linéaire.
$O(n)$	complexité linéaire (liée à la taille n des données),
$O(logn)$	complexité logarithmique (en base 2).
$O(1)$	en temps constant, indépendamment des entrées de l'algo.

On compare les algorithmes sur la base de leur complexité. La performance ou l'efficacité d'un algorithme sont des notions liées à la classe de complexité de l'algorithme.

Terminologie

- complexité spatiale : coût en quantité d'espace mémoire
- complexité temporelle : coût en rapidité d'exécution
- **grand O (O)** : comportement asymptotique (taille des données très grande, aux abords de l'infini). O donne un ensemble de majorants d'une fonction.
  thêta (Θ) : comportement dans les cas pratiques ; Θ donne une mesure plus juste intégrant minorants et majorants.
  petit o (o) : comportement dans les cas favorables ; donne les minorants.

Analyse de la complexité

L'**analyse de la complexité** fournit des outils et des mesures pour comparer des algorithmes du point de vue de leur efficacité ou performance. C'est cela que nous allons étudier dans cet enseignement, en même temps que la conception rigoureuse des algorithmes et donc des programmes ou logiciels ou applications (logicielles).

Il ne faut pas confondre l'analyse de la complexité avec la **théorie de la complexité** qui s'attache elle à la calculabilité, et l'étude des classes de problèmes, **solubles ou non solubles**...

2 Outils de base d'étude des algorithmes et de mesure de leur coût

AC21.03	✓	AC22.01		AC22.02		AC22.04	✓

Exercice 0 : ordres de grandeur

En Informatique, on se base sur les ordres de grandeur pour évaluer ou **estimer les coûts** (en ressources temps, espace, énergie, ...) des **algorithmes** (ne pas confondre avec programme).

Les coûts prennent la forme d'une fonction (disons f), qui dépend par exemple de la taille n d'une variable de l'algorithme ; n est par exemple la taille d'un tableau, le nombre de nœuds dans un arbre, le nombre d'arêtes, etc. mais n peut être aussi des opérations effectuées dans l'algorithme par exemple, des accès au réseau, des requêtes à une base de donnée, des rafraîchissements d'une page web, etc.

Exemples de fonctions :

$$f_1(n) = n^2 + 7$$
$$f_2(u) = u(u+1)/2$$
$$f_3(n) = n^4 + n^3(n-7)$$
$$f_4(m) = C.m.log(m) \text{ où C est une constante}$$

1. Comparez rigoureusement, sans entrer dans les détails, les fonctions f_1, f_2, f_3 et f_4 lorsque n est très grand.
2. Quels sont les ordres de grandeur de ces fonctions ?
3. Les fonctions f_1, f_2, f_3 et f_4 sont-elles linéaire, quadratique, exponentielle, logarithmique ? argumentez.
4. Intuitivement, lorsque vous faites le parcours d'un tableau de taille n avec deux boucles imbriquées l'une dans l'autre, quelle est la fonction (représentative) de coût de votre parcours ?
5. Lorsque n est très très grand (en millions, milliards, 10^6, 10^9, etc), comment comparez-vous les valeurs estimées de $n^2 + 2n + 1$ et $n^2 + 13$?
6. Soit un algorithme qui traite en un seul parcours un milliard d'objets ; le temps de traitement de chaque objet étant estimé à 1sec., à combien d'heures, de jours estimez-vous le traitement total ?
7. Comment estimez-vous le temps de l'algorithme précédent s'il devait comparer chaque objet à tous les autres objets ?

Exercice 1 : calculs élémentaires de coût

Soit l'algorithme présenté dans Algorithme 1.

1. Combien de **comparaisons d'éléments** de type $\mathbb{N}$, effectue-t-on lorsqu'on déroule cet algorithme ?
2. Quel est l'ordre de grandeur de ce nombre de comparaisons ?
3. Comparer les **coûts en comparaisons et en affectations d'éléments** de type $\mathbb{N}$ de l'algorithme Algorithme 1, avec ceux de Algorithme 2, page 8.
4. Considérons maintenant l'algorithme Algorithme 1 transformé comme dans **Algorithme 3**, page 9. Que dire des coûts de comparaisons ?
   dans quelle mesure peut-on améliorer l'algorithme comme dans la question 3 ?
5. Conclure (sur les coûts).

Algorithme 1 Un exemple d'algorithme pour le calcul de coût

1: **variables**
2: $i \in \mathbb{N}$ et $n \in \mathbb{N}_1$ et $n = \cdots$ et $M \in \mathbb{N}$ et $M = ...$ {N et M sont des constantes entières}
3: $cptP \in \mathbb{N}$ et $cptG \in \mathbb{N}$ {compteur des petits et des grands}
4: **fin variables**
5: $i \leftarrow 1$
6: $cptG \leftarrow 0$ {compteur des i plus grands que M}
7: $cptP \leftarrow 0$ {compteur des i plus petits ou égaux à M}
8: **tantque** $i \leq n$ **faire**
9: **si** $i \leq M$ **alors**
10: $cptP \leftarrow cptP + 1$
11: **sinon**
12: $cptG \leftarrow cptG + 1$
13: **fin si**
14: $i \leftarrow i + 1$
15: **fin tantque**
16: ecrire(cptP, cptG)

Algorithme 2 Un exemple d'algorithme pour le calcul de coût

1: **variables**
2: $i \in \mathbb{N}$ et $n \in \mathbb{N}_1$ et $n = \cdots$ et $M \in \mathbb{N}$ et $M = ...$ {N et M sont des constantes entières}
3: $cptP \in \mathbb{N}$ et $cptG \in \mathbb{N}$ {compteur des petits et des grands}
4: **fin variables**
5: $i \leftarrow 1$
6: $cptG \leftarrow 0$ {compteur des i plus grands que M}
7: $cptP \leftarrow 0$ {compteur des i plus petits ou égaux à M}
8: **tantque** $i \leq n$ **faire**
9: **si** $i \leq M$ **alors**
10: $cptP \leftarrow cptP + 1$
11: **fin si**
12: $i \leftarrow i + 1$
13: **fin tantque**
14: $cptG \leftarrow n - cptP$
15: ecrire(cptP, cptG)

Algorithme 3 Un exemple d'algorithme pour le calcul de coût (Tableau)

```
1: variables
2:     i ∈ ℕ  et  n ∈ ℕ1  et  n = ···  et  M ∈ ℕ  et  M = ... {N et M sont des constantes entières}
3:     T ∈ 1..n → ℕ  ∧  T = [...] {un tableau d'entiers}
4:     cptP ∈ ℕ  et  cptG ∈ ℕ {compteur des petits et des grands}
5: fin variables
6: i ← 1
7: cptG ← 0 {compteur des i plus grands que M}
8: cptP ← 0 {compteur des i plus petits ou égaux à M}
9: tantque i ≤ n faire
10:    si T[i] ≤ M alors {Modification de l'algo ici}
11:        cptP ← cptP + 1
12:    sinon
13:        cptG ← cptG + 1
14:    fin si
15:    i ← i + 1
16: fin tantque
17: ecrire(cptP, cptG)
```

Exercice 2 : calcul de coût

Nous considérons ici l'algorithme **Algorithme 4**, page 9.

Algorithme 4 Un exemple d'algorithme pour le calcul de coût d'accès à un Tableau

```
1: variables
2:     i ∈ ℕ  et  N ∈ ℕ1  et  N = ···
3:     j ∈ ℕ  et  M ∈ ℕ1  et  M = ···
4:     T ∈ 1..N × 1..M → ℕ  et  T = [···] {decl et initialisation du tableau T}
5: fin variables
6: pour i ∈ 1..N faire
7:     pour j ∈ 1..M faire
8:         T[i, j] ← i + j
9:     fin pour
10: fin pour
```

1. **Combien d'accès au tableau T** effectue-t-on (en dehors de l'initialisation) ?
2. Combien de **sommes d'entiers** effectue-t-on ?
3. Si on implante cet algorithme par un programme (avec les mêmes variables) sur une machine où **un entier est codé sur 64 bits**, quel est l'espace mémoire utilisé par le tableau de l'algorithme 4, page 9 ?
4. Quel est l'espace total utilisé par toutes les variables du programme .

Exercice 3 : Chasser les mauvaises pratiques

Évaluation des expressions (parfois coûteuses) plus d'une fois, ou dans les boucles.

Exemple de programmes trop souvent vus :

```
//        (A)
// en-tête  du programme
// . . .
maListe = ... // une liste donnée
i = ...
while (i < longueur(maListe)) {
  ...
  i=i+1
}
```

```
//             (B)
// en-tête  du programme
i = 1 ;     j = 1 ;
coteAdj = ... ; theta = ... ;
while (i <= N) {
   j=1
   while (j <= M) & (i*j < coteAdj/cos(theta)) {
     ...
     j=j+1
   }
   i=i+1
}
```

En argumentant les défauts de ces morceaux de programmes (A) et (B) relativement à l'expression des conditions de boucle, proposez une amélioration du coût de l'évaluation de chacun des programmes.

Exercice 4 : analyse des coûts des entrées/sorties

Les entrées/sorties sont généralement des opérations coûteuses (en temps), mais aussi en mémoire (elles passent par des *buffers* temporaires). Affichage progressif de données ou leur écriture progressive sur un périphérique peut devenir très coûteux lorsqu'on effectue des traitements itératifs. Il convient donc de bien analyser les situations pendant la conception des programmes.

Le pseudo-programme suivant (Algorithme 5) effectue progressivement des entrée-sorties sur un périphérique externe.

Algorithme 5 Un exemple d'algorithme avec des entrées/sorties

```
1: variables
2:     ··· {En-tête du programme}
3: fin variables
4: ···
5: tantque conditionB1 faire
6:     ···
7:     tantque condition B2 faire
8:         ···
9:         ··· calcul  donnees
10:        ecrire(peripherique, donnees)
11:    fin tantque
12:    ···
13: fin tantque
```

1. Après avoir analyser l'algorithme (Algorithme 5), proposer une amélioration de cet algorithme afin de diminuer son coût en temps, coût lié aux entrées-sorties.

2. quel est l'impact de la solution proposée sur les autres paramètres de coût d'un programme ? (imaginez aussi le cas des données de taille importante - très très très grande taille)

3. Discussion : équilibrage entre temps et mémoire.
   Dans la pratique, les applications nécessitant de nombreuses interactions avec les périphériques

ou avec le réseau (internet), par exemple l'envoi progressif de données versus envoi de données consolidées, les interactions avec un serveur de données, les publications progressives de données versus les publications massives, etc, nécessitent une analyse-conception rigoureuse en amont, pour garantir des résultats de coûts acceptables (y compris pour l'ergonomie).

Il est donc primordial d'étudier le compromis temps/mémoire et régler convenablement les paramètres des programmes.

Ouverture - sensibilisation - approfondissement - recherche (travail personnel)

1. Travail collaboratif sur des documents partagés/synchronisés via le réseau/dans le cloud, avec des serveurs et des *datacenters* refroidis (à grand frais)...
2. Éditeurs collaboratifs : trop coûteux ?!
   Un document en édition collaborative est souvent enregistré sur un *drive*, donc sur un serveur distant. A chaque modification (même d'un seul caractère), le document est mis à jour (à travers le réseau !). Il y a donc des interactions continues via le réseau, lors de l'édition d'un document collaboratif.
3. Gestionnaires de versions avec synchronisation à la demande : moins coûteux que les éditeurs collaboratifs.

   On édite les documents en local, et au besoin on les met à jour pour les partager avec les aux autres utilisateurs du document. Exemples svn, git, mercurial
4. Réflexions sur le bon usage des outils collaboratifs interactifs tels que `XXXdrive, UNCloud, googledocs, discord, bbb`, etc
5. Sensibilisation coût en temps+mémoire des algorithmes d'exploitation de masses de données (*big data*).

3 Eléments de construction systématique de programmes

AC21.03	✓	AC22.01		AC22.02	✓	AC22.04	

Exercice 1 : étude du coût algorithmique de la recherche d'un mot dans un dictionnaire.

Supposons que nous voulions rechercher un mot x dans un dictionnaire DICO ; on l'y trouvera ou non. Appelons taille des données, le nombre de mots dans notre dictionnaire, et convenons que l'opération élémentaire est la comparaison de deux mots (le mot x avec un mot courant du dictionnaire).

On peut, avec un premier **algorithme naïf**, parcourir les mots du dictionnaire, du début à la fin, dans l'ordre, jusqu'à la terminaison du processus. Cette terminaison peut alors prendre deux formes : soit le mot x est trouvé avec un nombre de comparaisons inférieur ou égal à n, soit le mot n'est pas trouvé, avec n comparaisons effectuées.

On peut distinguer ce dernier cas, en utilisant la ***technique de la sentinelle*** qui consiste à ajouter le mot cherché en fin de dictionnaire, donc à la place $n + 1$. Ainsi, on trouve toujours le mot mais, avec $n + 1$ comparaisons on sait que le mot n'était pas dans le dictionnaire.

Sur ce principe, on propose l'algorithme naïf suivant (Algorithme 6, page 12), pour en étudier le coût. On suppose en entrée de l'algorithme, le dictionnaire sous la forme de d'un tableau $DICO[1..n]$ de mots différents, classés par ordre lexicographique croissant, et un mot x.

Algorithme 6 Algorithme RechDic1

```
1:  constants
2:      n ∈ ℕ et n = ··· et x ∈ chaineDeCarac et x = ···  { n et x sont des données de l'algo}
3:  fin constants
4:  variables
5:      DICO ∈ 1..n + 1 → chaineDeCarac {dictionnaire représenté comme un tableau de chaine de caract.}
6:  fin variables
7:  début
8:      DICO[n + 1] ← x; {sentinelle}
9:      i ← 1;
10:     tantque DICO[i] ≠ x faire
11:         i ← i + 1
12:     fin tantque
13:     si i < n + 1 alors
14:         écrire(oui! mot trouvé)
15:     sinon
16:         écrire(non!, mot non trouvé)
17:     fin si
18: fin début
```

En considérant l'opération élémentaire de comparaison de mots, et la complexité en termes du nombre de cette opération élémentaire,

1. quel est le coût (ou la complexité) dans le **meilleur des cas** ? explicitez le cas
2. quel est le coût (ou la complexité) dans le **pire des cas** ? explicitez le cas
3. quel est le coût (ou la complexité) **pratique** de cet algorithme ? explicitez la situation.

Puisqu'un dictionnaire est trié, il existe une méthode plus rapide que l'algorithme naïf, pour y chercher un mot : la recherche dichotomique (dont il existe plusieurs variantes).

Etude d'un algorithme de meilleur coût : recherche dichotomique. Etudions à présent une variante itérative (Algorithme 7, page 13) de l'algorithme de recherche dichotomique.

Principe de l'algorithme de recherche dichotomique : on compare le mot x cherché avec le mot situé en milieu du dictionnaire (coupant ainsi le dictionnaire en deux parties) ;
- soit le mot x est plus grand (au sens lexicographique) que le mot au milieu et on poursuit la recherche de la même façon dans la partie 'supérieure' du dictionnaire (en prenant son milieu, etc) ;
- soit le mot x est plus petit que le mot au milieu et on poursuit la recherche dans la la partie 'inférieure' ;
- soit il est égal et on s'arrête.

Ce processus de décomposition du dictionnaire en deux, à chaque étape est poursuivi jusqu'à trouver, ou pas, le mot cherché.

Algorithme 7 Algorithme RechDicDichotom

```
1:  constants
2:      n ∈ ℕ et n = ··· et x ∈ chaineDeCarac et x = ···
3:  fin constants
4:  variables
5:      DICO ∈ 1..n → chaineDeCarac et DICO = [...]
6:      deb ∈ ℕ1 et fin ∈ ℕ1 et fin ≥ deb et mil ∈ ℕ1 et mil ≥ deb et mil ≤ fin
7:  fin variables
8:  début
9:      deb ← 1; fin ← n;
10:     tantque (deb ≠ fin) faire
11:         mil ← (deb+fin)/2;
12:         si (x > DICO[mil]) alors
13:             deb ← mil + 1
14:         sinon
15:             fin ← mil
16:         fin si
17:     fin tantque
18:     si (DICO[deb] = x) alors
19:         écrire(oui! mot trouvé, position, deb)
20:     sinon
21:         écrire(non! mot non trouvé dans le dico)
22:     fin si
23: fin début
```

4. comparer les coûts des deux algorithmes présentés, en fonction du nombre de comparaisons de mots.

Exercice 2 : découverte de schéma de récursion

Montrez que la somme des n premiers entiers impairs est égale à n^2. C'est à dire

$$\sum_{i=1}^{n}(2i-1) = n^2$$

Exercice 2' : schéma de récursion - calcul du nombre d'arbres binaires de taille n

En construction de programmes, de nombreux problèmes (par exemple, ceux liés au dénombrement, au calcul de valeur optimale d'une fonction, etc), se résolvent en mettant en évidence une récurrence permettant le calcul de la grandeur voulue. Il importe que la récurrence soit complète (en envisageant toutes les situations possibles).

Pour illustrer, on veut calculer le nombre nbab(n) d'arbres binaires ayant n nœuds (les feuilles comprises). Pour cela, on va d'abord établir la récurrence définissant ce nombre.

Analyse et systématisation

On remarque qu'il y a un seul arbre n'ayant aucun élément : l'arbre vide.

Pour définir une récurrence sur nbab(n), on considère la décomposition d'un arbre binaire à n nœuds en arbres plus petits. Pour n positif, un arbre à n nœuds est constitué d'une racine (un nœud), éventuellement d'un sous-arbre gauche ayant i nœuds, et éventuellement d'un sous-arbre droit ayant (n-i-1) nœuds.

Etude et construction

1. De combien de façons différentes peut-on placer un arbre vide (zéro nœud) ?
2. De combien de façons différentes peut-on placer un nœud à gauche (d'une racine) ?
3. De combien de façons différentes peut-on placer 2 nœuds à gauche ?
   Proposez un schéma, pour 2 et pour 3 nœuds.
   Pour chacune, de combien de façon peut-on placer le reste des nœuds à droite, si on devait placer n nœuds ?
4. De combien de façon peut-on placer i nœuds à gauche ?
   pour chacune, de combien de façon peut-on placer le reste des n nœuds à droite ?
5. Finalement comment peut-on calculer le nombre d'arbres binaires ayant n nœuds ?
6. En déduire une fonction récursive de calcul de nbab(n).

D'un point de vue algorithmique, on procède au calcul en construisant un programme itératif dans lequel **un tableau** à une ou plusieurs dimensions permet de stocker la succession des valeurs de la suite calculées. On calcule ainsi la valeur d'**une cellule** du tableau à partir de celles déjà présentes dans le tableau.

L'ordre de remplissage du tableau est primordial et la progression du calcul dépend des relations de dépendance entre les éléments (c'est plus aisé avec une récurrence à un indice, qu'à plusieurs indices).

7. Vérifier avec nbab(3), que le calcul ne dépend que de valeurs précédentes calculées.
8. Construire, en utilisant un tableau pour stocker les valeurs déjà calculées, un algorithme itératif pour effectuer le calcul de nbab(n).

Exercice 3 : nombre de partitions en p blocs d'un ensemble de n éléments

On note $\mathcal{S}(p,n)$ le nombre de partitions à p blocs d'un ensemble à n éléments. Les valeurs $\mathcal{S}(p,n)$ sont appelées nombre de *Stirling*. Par exemple, $\{\{a,c\},\{b,d\}\}$ et $\{\{b\},\{a,c,d\}\}$ sont deux des sept partitions à deux blocs de l'ensemble D à quatre éléments $\{a,d,b,c\}$.

1. Pour n = 4 éléments, donnez toutes les partitions à un, deux, trois et quatre éléments (p=1, p=2, p=3, p=4) de cet ensemble D (sachant qu'un bloc ne peut pas être vide)
2. Donnez maintenant pour p = 2, les partitions de deux éléments (p=2) et leur nombre, d'un ensemble D à un, deux, trois, quatre éléments (on fait varier n ; n=1, n=2, n=3, n=4).
3. Quelle est la relation de récurrence définissant $\mathcal{S}(p,n)$?
4. Proposez une progression du calcul de $\mathcal{S}(p,n)$ puis l'algorithme itératif correspondant, en choisissant la structure de données adaptée.
5. Donner la valeur de $\mathcal{S}(p,n)$ pour $1 \leq p \leq 5$ et $1 \leq n \leq 7$.

Exercice 4 : algorithme de recherche d'un gué dans le brouillard

Il s'agit de l'abstraction d'un problème courant qui montre comment des stratégies analogues peuvent produire des algorithmes de complexités différentes. Voici l'énoncé du problème.

Vous êtes devant une rivière, dans un épais brouillard. Vous savez qu'il y a un gué (Fig. 1) dans les parages, mais vous ignorez s'il est à gauche ou à droite, et à quelle distance. Avec ce brouillard, vous verrez l'entrée du gué seulement quand vous serez juste devant. Comment faire pour traverser la rivière ?

Figure 1: Un passage à gué (wikipédia)

On propose la stratégie suivante : il faut explorer successivement à droite, à gauche, à droite, etc. en augmentant à chaque fois la distance. Commencer par la gauche ou par la droite n'a pas d'importance, mais il faut traiter les deux côtés de manière "équilibrée" et augmenter la distance régulièrement afin d'éviter de faire trop de chemin du côté où le passage ne se trouve pas.

Une première méthode consiste à faire un pas à droite, revenir au point de départ, un pas à gauche, revenir au point de départ, deux pas à droite, revenir au point de départ, deux pas à gauche, revenir au point de départ, trois pas à droite, et ainsi de suite jusqu'au succès.

1. Faites un schéma synthétique de la stratégie, pour l'observer attentivement ; esquissez ensuite le schéma de calcul du nombre de pas.
2. en supposant que le gué se trouve **à gauche à 15 pas**, quel est le nombre de pas effectués pour y arriver ?

3. donnez dans le cas général (gué à distance de n pas), le nombre de pas effectués avec la méthode proposée si le **gué est situé à n pas à gauche** (resp. à droite) du point de départ.
   Quelle est la classe de complexité de cette méthode en terme de nombre de pas ?

On souhaite maintenant trouver une méthode fondamentalement plus rapide (au sens de la classe de complexité) permettant de garantir que le gué est trouvé en moins de 10*n pas. Le défaut de la précédente réside dans une progression trop 'prudente'. Augmenter d'un pas à chaque fois (c'est une progression arithmétique) coûte finalement cher et on envisage de doubler le nombre de pas à chaque passage (une progression géométrique).

4. Esquissez un schéma de cette variante de la stratégie.
5. Donnez le nombre de pas effectués selon que le gué se trouve à 9 pas (resp. 15 pas) à gauche ou à droite du point de départ.
6. Conclure : quelles conclusions générales peut-on tirer de la méthode et des 4 résultats (nombre de pas) calculés ?
7. La seconde méthode a permis de faire mieux que la première, mais elle reste un peu "rustique". Modifier la pour atteindre l'objectif visé, à savoir un nombre de pas inférieur à 10*n

4 Spécification et construction de programmes

AC21.03	✓	AC22.01	✓	AC22.02	✓	AC22.04	

mots-clés : Spécification Pre/Post - Terminaison - Triplet de Hoare - invariant

Construction correcte des itérations ou boucles

Rappel du schéma de construction des boucles Construire rationnellement un algorithme ou un programme consiste à le voir comme un mécanisme faisant passer un "système" d'une situation initiale appelée précondition à une situation finale nommée "postcondition" ou "but".

Le couple (précondition, postcondition) est appelé la *spécification* du programme.

On cherche à construire un programme à partir de sa *spécification* (voir votre cours : *spécification* ≡ *(precondition, postcondition)*).

On rappelle que la construction de boucle est systématisée par les constituants suivants :

1. **Invariant** (de boucle) : un prédicat représentant une propriété caractéristique du problème traité.
2. **Condition d'arrêt** : C'est également un prédicat, qui exprime la condition d'arrêt de la boucle
3. **Progression** : c'est une action/instruction ou un bloc d'actions.
4. **Initialisation** : c'est une action/instruction ou un bloc d'actions, effectué une seule fois au début.
5. **Expression de Terminaison** : c'est une expression à valeur entière (qui va croître tout le temps ou bien décroître tout le temps).

Les relations suivantes réunissent ces constituants :

R1 La conjonction de l'invariant et de la condition d'arrêt conduit au but recherché par le programme :

$$(\text{Invariant et condition d'arrêt}) \Rightarrow \text{post-condition}$$

R2 la progression doit

(a) conserver l'invariant. Avant l'exécution de la progression, le prédicat invariant et non condition d'arrêt est vrai. Avant et après l'exécution de la progression, l'invariant doit être vrai.

(b) faire décroître strictement l'expression de terminaison. Ainsi à chaque tour de boucle, l'expression est positive ou nulle, et strictement inférieure à sa valeur précédente ; la condition d'arrêt sera donc atteinte au bout d'un temps fini. Cela permet de garantir que le programme construit *se termine* (donc pas de boucles infinies).

R3 L'initialisation doit instaurer l'invariant. Notons que la pré-condition de l'initialisation est la pré-condition du programme et sa post-condition est l'invariant de la boucle.

A travers les relations **R1, R2, R3**, on note que l'invariant est un constituant fondamental pour la conception/construction des boucles. Ainsi il est raisonnable de commencer la construction de boucle par l'identification de l'invariant.

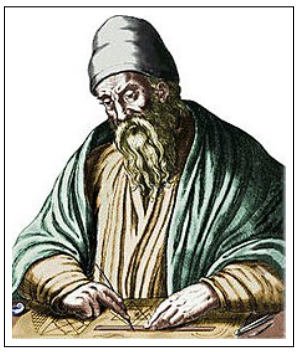


Figure 2: Euclide (wikipédia)

Exercice 1 : division euclidienne

Encore Euclide (d'Alexandrie) ! qui est-ce ? `https://fr.wikipedia.org/wiki/Euclide`

Considérons ici le problème classique de la division euclidienne. On cherche le quotient de la division "entière" de l'entier A par l'entier B ; (q,r = divisionEuclide(A,B)) ; étant entendu que l'on ne dispose pas de l'opération de division.

1. Donner la **spécification** du programme de division euclidienne
2. Proposer un invariant pour la construction (le calcul), en vous faisant au préalable une bonne idée du traitement à faire (= analyse).
3. Proposer une condition d'arrêt du calcul.
4. Proposer une progression de calcul.
5. Proposer une initialisation de la construction.
6. En déduire, de tout ce qui précède, la construction finale de l'algorithme de calcul de la division.

Exercice 2 : construction du drapeau monégasque (méthode du *travail réalisé en partie*)

Il s'agit d'un problème de rangement d'objets selon leurs catégories ou propriétés. Une abstraction en est donnée par l'exercice du drapeau monégasque (Fig. 3), classique en algorithme.

Figure 3: Le drapeau 'rouge-blanc' monégasque (wikipédia)

La situation initiale est un tableau T[1..N] avec $N \geq 0$, dont les éléments sont de couleur blanche ou rouge et forment le sac S. (Rappel : un sac est une structure de données contenant des objets en vrac, et des objets peuvent être en plusieurs exemplaires). La situation finale est un tableau formant le même sac S, dans lequel les éléments blancs occupent la partie gauche et les éléments rouges la partie droite. On s'impose de passer de la situation initiale à la situation finale en une seule boucle, qui parcourt le tableau de gauche à droite. Les seuls changements du tableau s'effectuent par l'opération d'échange des éléments d'indice i et j de T notée echangerT(i,j), ce qui garantit la préservation du sac de valeurs S initial.

Pour aider à trouver un invariant, on admet qu'à un moment où *le travail est partiellement réalisé*, on a atteint la configuration décrite par la figure Figure 4. C'est à dire qu'on progresse dans le traitement en poussant à gauche tous les éléments blancs, et à droite tous les éléments rouges ; ainsi, il y a une zone non encore traitée dans laquelle on ne peut rien dire, quand on n'a pas atteint la fin du traitement.

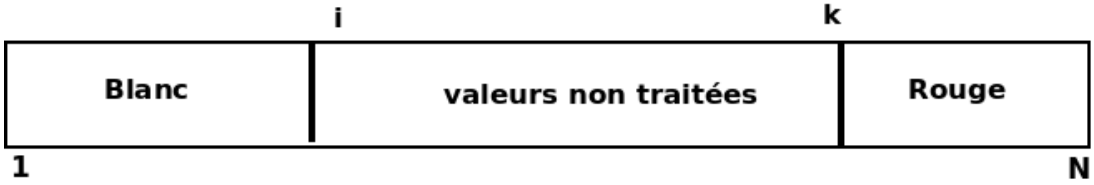


Figure 4: Configuration du drapeau monégasque partiellement traité

1. Sur la base de la figure Fig. 4, proposer un invariant qui caractérise le traitement partiel effectué.
2. Expliciter, tous les cas couverts par votre invariant temporaire ;
   (en fonction des configurations possibles de la donnée d'entrée (le contenu tableau)).

   En Supposons que cet invariant temporaire ferait l'affaire, ou sinon serait amélioré,
3. Proposer une condition d'arrêt de la construction.
4. Proposer la progression de la construction.
5. Proposer une initialisation.
6. Proposer l'expression de terminaison.
7. Déduire l'algorithme complet pour la construction du drapeau.

Exercice 3 : Recherche dans un tableau à deux dimensions à l'aide d'une seule boucle

On étudie une méthode pour rechercher un élément dans un tableau à deux dimensions, en ne faisant qu'une seule boucle, au lieu de la solution ordinaire qui utilise deux boucles imbriquées.

Soit un tableau avec L lignes et C colonnes contenant des entiers naturels, et soit V un entier à rechercher dans T. On fait l'hypothèse que T contient au moins une ligne et une colonne et T contient V. On cherche (i,j) telle T[i,j] contient V.

1. Proposer une spécification de l'algorithme à écrire : c'est-à-dire une précondition prec et une post-condition postc, telles que on aura
$$\{prec\}\ \ algoRech\ \ \{postc\}$$
2. En utilisant la technique du *travail en partie réalisé*, suggérer la façon d'avancer dans le tableau en faisant un seul parcours/boucle.
3. Proposer les 5 constituants (**invariant, condition d'arrêt, progression, initialisation, terminaison**) d'une boucle unique qui répond à la spécification.
4. En déduire l'algorithme de recherche et préciser sa complexité en termes de nombre de comparaisons d'éléments.

Exercice 4 : Construction d'un algorithme de tri par sélection

Cet algorithme de tri n'est pas parmi les plus rapides ; nous l'étudions pour renforcer la pratique de la construction systématique des itérations.

On donne la spécification suivante:

- **Pre-condition** : $T : 1..N \to \mathbb{N}$ et S un sac des valeurs contenues dans T
- **Post-condition** : T est trié par ordre croissant :
$$T : 1..N \to \mathbb{N} \quad \text{et} \quad \forall i.((i \in 1..N-1) \Rightarrow T[i] \leq T[i+1])$$

On veut construire un programme de tri par sélection satisfaisant cette spécification ; on utilise aussi *la méthode du travail partiellement réalisé*. C'est à dire qu'à un moment donné pendant le déroulement de l'algorithme, une partie du tableau est bien triée, et le reste n'est pas encore traité. Par exemple la partie en début du tableau est triée jusqu'à un indice et la suite ne l'est pas pour sûr.

Deux options sont possibles, lorsqu'on est à un indice i : soit T[i] sera inséré à sa place en début de tableau (c'est le *tri par insertion*), soit T[i] sera échangé avec un élément plus petit que lui dans le reste du tableau (c'est *le tri par sélection*), mais alors il faut être sûr que le reste du tableau ne contient que des éléments plus grands que ceux déjà traités et rangés au début du tableau.

Nous proposons d'**utiliser ici la deuxième option**. Nous supposons aussi définie une procédure echanger(i,j) qui échange les éléments de T d'indices respectifs i et j. Les indices i et j se situent entre $1..N$.

1. Donner la post-condition associée au tri par sélection (avec l'hypothèse d'une partie du travail réalisé).
2. Les seules modifications de T sont celles effectuées par la procédure echange(i,j) qui échange les éléments qui se trouvent aux positions i et j dans le tableau. **Quelle hypothèse devrons nous faire** pour garantir que le tableau T a exactement les mêmes éléments en entrée qu'en sortie de l'algorithme (c'est-à-dire que le sac S est le même en sortie de l'algorithme) ?
peut-on écrire la spécification de cette procédure ?

3. Donner les 5 constituants (**invariant, condition d'arrêt, progression, initialisation, terminaison**) de la boucle ; on néglige pour le moment la recherche du minimum dans la partie non encore traitée.

4. Sélection de l'élément minimum. Construire de la même façon une boucle auxiliaire pour trouver la position m du minimum du tableau T[i..N].

   Exprimer la spécification de cette boucle auxiliaire.

5. Construire la boucle auxiliaire.

6. Enfin, construire l'algorithme de tri par sélection.

7. Quelle est la complexité de l'algorithme en nombre d'échanges ?

Exercice 5 : Algorithme du drapeau hollandais (de Dijkstra)

L'algorithme que nous allons étudier est dû à E. W. Dijkstra. Il s'agit de la reconstitution des couleurs (rouge, blanc, bleu) de la Hollande (Fig. 5, page 21), pays dont est originaire son auteur. Cet algorithme est à la base du célèbre algorithme de tri rapide dit *Quicksort*.

Dans l'étude suivante, les couleurs sont remplacées par des entiers naturels.

Figure 5: Le drapeau 'rouge-blanc-bleu' hollandais, (wikipedia)

On donne la spécification suivante de l'algorithme à construire :

- **Pre-condition** : T est un tableau de N entiers naturels et S un sac des valeurs contenues dans T avec $N \geq 1$ et $V = T[1]$ et
- **Post-Condition** : S est le sac des valeurs contenues dans T et T se compose de trois (3) parties : à gauche des valeurs inférieures à V, à droite des valeurs supérieures à V et au centre, toutes les occurrences de la valeur V.

Schématiquement, nous avons la situation dans la figure Fig.6, page 22 :

Dans la suite, pour réaliser l'algorithme et donc le traitement prévu, on va parcourir le tableau T et échanger les valeurs situées à des indices (deux indices) afin de les mettre à leur bonne place par rapport à l'état courant du tableau (comme nous l'avions fait dans l'exercice du tri par sélection).

On suppose donnée une procédure echange(i,j), qui échange les éléments du tableau qui se trouvent aux positions i et j dans le tableau. Les indices i et j se situent entre 1..N. Les seules modifications de T sont celles effectuées par la procédure echange(i,j).

1. On voudrait que le sac S des valeurs contenues dans T, soit le même à la fin qu'à l'entrée de l'algorithme. Quelle(s) condition(s) doit remplir la procédure echange(i,j), ou quelle hypothèse doit-on y faire ?

Figure 6: Configuration du drapeau hollandais

2. Exprimer formellement (en Logique) la post-condition pour son utilisation dans la construction de l'algorithme.

3. Proposer un invariant pour la construction de l'algorithme. On utilisera la *technique du travail partiellement réalisé*. On introduira si besoin une nouvelle variable en plus de b, w et N ; et on pourra modifier si besoin la post-condition.

4. En analysant les configurations possibles des entiers dans le tableau, identifier les configurations qui ne nécessitent pas d'échanger les éléments du tableau.

5. Proposez les 4 constituants restants (**condition d'arrêt, progression, initialisation, terminaison**) d'une boucle qui répond à la spécification donnée pour l'algorithme

6. En fonction des constituants précédemment construits, donner l'algorithme final.

7. Etudier la complexité de l'algorithme, en terme de nombre d'échanges effectués.

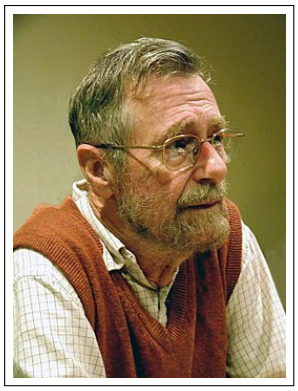
5 Exploitation de structures de données

Diviser pour régner - algorithmes gloutons - graphes

AC21.03	✓	AC22.01	✓	AC22.02	✓	AC22.04	

Etude de cas 1 : Algorithmes de Prim et Dijkstra

R. C. Prim
(www.ithistory.org/honor-roll/dr-robert-clay-prim)



E. W. Dijkstra
(fr.wikipedia.org/wiki/Edsger_Dijkstra)

On considère des graphes ou réseaux de nœuds qui sont connectés de façon bidirectionnelle ou bien de façon unidirectionnelle (orientée).

Un premier problème dont la solution est apportée par l'algorithme de Prim est basée sur un graphe avec des liens bidirectionnels (voir Fig. 7, page 23) ; chaque lien a un coût qui est un entier positif. Tout nœud (ou site) peut atteindre tout autre nœud en empruntant une suite de liens. On recherche un acheminement d'une information entre deux nœuds, tel que **la somme des coûts des liens empruntés pour atteindre l'ensemble des sites soit minimale**.

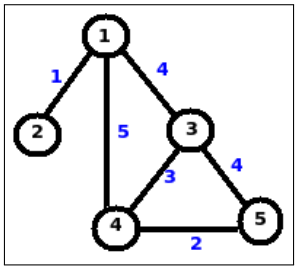


Figure 7: Un graphe non oriente, connexe

Un deuxième problème dont la solution est apportée par l'algorithme de Dijkstra, les nœuds (ou sites) sont reliés par des liens orientés (unidirectionnels, voir Fig 8, page 24), et il se peut qu'il n'existe pas de liens entre un couple de nœuds du graphe. La problématique est la même que dans le problème précédent, seulement ici, on recherche un acheminement dont l'**optimalité porte sur la somme des coûts associés aux liens empruntés pour atteindre chaque site.**

Il existe de nombreux problèmes pratiques résolus sur la base de ces algorithmes fondamentaux : câblage d'un ensemble de bâtiments, système optimal d'irrigation de parcelles, acheminements dans les réseaux, acheminements dans les transports, etc

Étude de l'algorithme de Prim Soit $G = (N, A, P)$ un graphe non orienté dont l'ensemble des sommets ou nœuds est appelé N et l'ensemble des *arêtes* appelé A ; G est valué par une fonction P (poids) sur les entiers

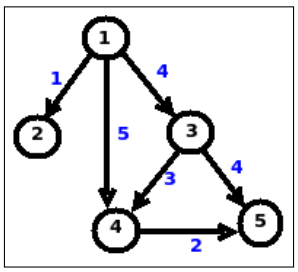


Figure 8: Un graphe unidirectionnel

naturels positifs ($P : A \rightarrow \mathbb{N}$) et est connexe (ie, il existe une chaîne reliant toute paire de sommets distincts de G). De façon analogue, $T = (N, A, P)$ désigne un arbre (*Tree*) non orienté composé de sommets (resp. arêtes) N (resp. A), valué sur les entiers positifs.

On appelle poids d'un arbre la somme des valeurs de ses arêtes (ou branches). Notons aussi qu'un arbre de k sommets possède (k-1) arêtes, d'autre part qu'une chaîne élémentaire unique relie toute paire de sommets distincts d'un arbre.

☛ On peut voir une animation de l'algorithme de Prim ici `https://www.pairform.fr/doc/1/32/180/web/co/Prim.html`

On appelle **arbre de recouvrement**[1] du graphe $G = (N, A, P)$, un arbre à priori non enraciné, incluant tous les sommets de G (voir Fig. 9, page 24). Un arbre de recouvrement de G de plus faible poids est appelé arbre de recouvrement de poids minimal (*arpm*) ou parfois arbre sous-tendant de poids minimal (*astm*). On sait montrer (**P1**) qu'un graphe G connexe, admet au moins un *arpm*, et qu'un arbre est son propre *arpm*.

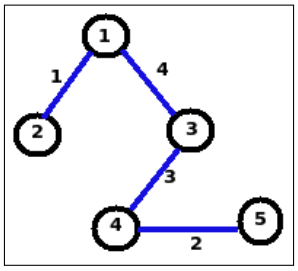


Figure 9: Un arbre de recouvrement associé au graphe de la fig. 8

En référence à un graphe $G = (N, A, P)$, on dit d'un arbre T qu'il est *prometteur* s'il est un sous-arbre d'un *arpm*.

On sait démontrer (**P2**) que l'adjonction à un arbre prometteur $T = (N', A', P')$ de $G = (N, A, P)$, de l'une des arêtes mixtes de G de valeur minimale, résulte à un nouvel arbre prometteur.

Construction de l'algorithme de Prim (comme un algorithme glouton exact[2]).

Principe de l'algorithme de Prim : on fixe un sommet comme source, sommet qui constitue un arbre prometteur (sommet numéro 1 dans la suite) ; ensuite on fait grossir l'arbre prometteur (la source initialement) en lui adjoignant une nouvelle arête (et donc un nouveau sommet aussi) du graphe G que l'on veut recouvrir. Un tel ajout ne peut se faire qu'avec une **arête mixte**, c'est à dire qui est constituée d'un sommet pré-existant (sommet interne) dans l'arbre, et d'un sommet n'en faisant pas partie (sommet externe). La question est donc de déterminer quelle arête de G peut/doit être ajoutée dans un arbre prometteur pour que le nouvel arbre résultant soit lui aussi prometteur.

[1] spaning tree

[2] c'est un algorithme qui consomme des ressources, et fait des choix sans revenir en arrière (backtrack)

On adopte le *principe de la course en tête*, pour construire l'algorithme.

Principe de la 'Course en tête'

Le principe est de considérer le critère d'optimalité dès le départ d'un algorithme qui procède par exemple à des choix successifs, étape par étape, pour la suite d'un traitement. Ainsi à chaque étape, le choix le plus optimal est effectué ; et ainsi de suite lors des étapes suivantes. Cela peut être lors du parcours d'un arbre/graphe pour le choix du noeud suivant, la tâche suivante lors de l'ordonnancement de tâches, etc

On propose la postcondition suivante pour l'algorithme à réaliser :
postcondition $T = (N', A', P')$ est un arbre prometteur de $G = (N, A, P)$ et $card(A') = card(N') - 1 = card(N) - 1 = n - 1$.

Première partie : construction d'une boucle du traitement principal (Algo. Prim)

1. En considérant la post-condition proposée et la technique du travail partiellement réalisé, proposez un **invariant** de l'algorithme.
2. Proposez une condition d'arrêt de l'algorithme.
3. Proposez une progression (en exploitant l'ajout d'arête (propriété **P2**).
4. Proposez une expression pour la terminaison.
5. Proposez une initialisation.

Deuxième partie : modélisation/structuration des données
Dans la suite nous allons manipuler des arêtes pour construire progressivement l'arbre prometteur T. Une arête relie un sommet (identifié par un entier, 1..n) à exactement un autre sommet (identifié aussi par un entier, 1..n).

1. Proposez une structure (de donnée) pour représenter les arêtes entre les sommets 1..n et 1..n
2. En considérant que le sommet source (identifié par 1), ne sera lié à aucun autre sommet interne (dans T lui même) contrairement aux autres sommets qui vont progressivement être liés à un sommet dans T, proposez une meilleure définition de la structure de donnée.

Troisième partie : modélisation de la progression des traitements
Dans la suite, nous représentons l'ensemble des arêtes mixtes par une fonction (aM) qui à chaque et à tout sommet externe (avec un identifiant **e** entre 2..n), associe le sommet interne (avec un identifiant **i** entre 1..n) le plus proche au sens de la valeur des arêtes reliant les sommets.

$$areteM \in 2..n \rightarrow 1..n$$

$$e \mapsto i$$

Ainsi, à chaque progression on devra mettre à jour la fonction *areteM* (car il y aura une arête de moins, qu'on aura ajoutée à l'arbre).

Parmi les nombreuses façons de représenter l'arbre résultant, en cohérence avec *areteM*, on choisit un arbre enraciné sur la source (le sommet 1). En conséquence, la fonction *areteI*, va représenter les arêtes qui font partie intégrante de l'arbre (en cours d'élaboration) en associant à tout sommet (sommet source exclu), son père dans cet arbre ($areteI \in 2..n \rightarrow 1..n$).

Tout sommet du graphe G est, à un moment, soit déjà dans l'arbre en construction, soit encore dans le graphe G ; l'ensemble des sommets externes et celui des sommets internes constituent donc une partition de l'ensemble N des sommets de G.

1. Puisque l'ensemble des sommets externes est exactement le *domaine* de la fonction *areteM* (notons le dom(areteM)), et celui des sommets internes est exactement le *domaine* de la fonction *areteI* (notons le dom(areteI)),
   Exprimer que dom(areteM) et dom(areteI) forment une partition de N. On s'en servivra dans l'algorithme en construction.

Puisque le graphe considéré est non orienté, on peut le représenter par une matrice P telle que $P = P^T$ où P^T est la matrice transposée de la matrice P. Dans ces deux matrices, l'absence d'arête dans le graphe se traduit par la valeur conventionnelle $+\infty$. Ci-dessous (Table 1) une matrice P correspondant au graphe non orienté de la figure Fig. 7, page 23.

P	1	2	3	4	5
1	$+\infty$	1	4	5	$+\infty$
2	1	$+\infty$	$+\infty$	$+\infty$	$+\infty$
3	4	$+\infty$	$+\infty$	3	4
4	5	$+\infty$	3	$+\infty$	2
5	$+\infty$	$+\infty$	4	2	$+\infty$

Table 1: Matrice du graphe de la figure Fig. 7

Avec ces considérations, on construit la version abstraite suivante (Algo. 8) de l'algorithme en vue :
Nous allons maintenant raffiner cet algorithme

1. Proposez une sructure de données pour représenter *areteI* et *areteM*
2. Ecrivez une fonction *SommetExterneCoutMin* pour réaliser la recherche du sommet externe le plus proche d'1 sommet externe i donné (voir les lignes 1X..1Y de l'algorithme.)

$$(e, i) \in areteM \quad \wedge \quad P(e, i) = min(\{P(l, i) \mid (l, i) \in areteM\})$$

3. Et quoi d'autres ?

Références

https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra
https://fr.wikipedia.org/wiki/Algorithme_de_Prim
https://en.wikipedia.org/wiki/Prim%27s_algorithm
https://www.ithistory.org/honor-roll/dr-robert-clay-prim

Algorithme 8 Algorithme de Prim - version abstraite

1: **constants**
2: $n \in \mathbb{N}_1$ et $n = \cdots$ et $P \in 1..n \times 1..n \to \mathbb{N}_1$ et
3: $P = P^T$ et $P = [\cdots]$ et $EstConnexe(G)$
4: **fin constants**
5: **variables**
6: $areteI \in 2..n \to 1..n$ et $areteM \in 2..n \to 1..n$ et
7: $dom(areteI) \cup dom(areteM) = 2..n$ et
8: $dom(areteI) \cap dom(areteM) = \emptyset$
9: **fin variables**
10: **début**
11: {Prec: $\cdots$}
12: $areteI \leftarrow \emptyset$;
13: $areteM \leftarrow 2..n \times \{1\}$; {Toutes les aretes ont 1 comme arête père}
14: **pour** $e \in 2..n$ **faire**
15: **soit** (e, i) **tel que** $(e, i) \in areteM$ et $P(e, i) = min(\{P(l, i) \mid (l, i) \in areteM\})$
16: {(e,i) est l'arête mixte de valeur minimale }
17: **début**
18: $areteI \leftarrow areteI \cup \{(e, i)\}$;
19: $areteM \leftarrow areteM - \{(e, i)\}$;
20: **pour** $e' \in dom(aM)$ **faire** {recherche e' sommet externe le plus proche du nouveau interne e}
21: **si** $(P(e', e) < P(e', areteM(e')))$ **alors**
22: $areteM(e') \leftarrow e$
23: **fin si**
24: **fin pour**
25: **fin début**
26: **fin pour**
27: {Postc : la postcondition}
28: écrire(résultat: T)
29: **fin début**