École Supérieure d'Ingénieurs de Poitiers
Gea 2

Algorithmique et Programmation

Laurent Signac

Table des matières

Avant Propos

Ce fascicule constitue le support de cours utilisé à l'École Supérieure d'Ingénieurs de Poitiers, dans la spécialité Génie Électrique et Automatique en deuxième année. Il fait suite au polycopié de tronc commun de première année, en ce sens que des problèmes d'algorithmique y sont encore traités. En revanche, une partie conséquente, et plus technique, est consacrée à la programmation en C.
Le lecteur se rendra vite compte que les sujets qui auraient pu être traités sont inépuisables. En conséquence, il est intéressant de pouvoir se reporter à des ouvrages plus spécialisés ou plus complets. En voici une liste (la bibliographie se trouve à la fin) :

- En ce qui concerne l'algorithmique en général, [4] constitue ne référence, aborde largement plus de sujet que ceux qui sont traités ici, et de manière très rigoureuse. Néanmoins, il ne contient aucune référence technique (de programmation), et fait appel à un langage algorithmque créé pour l'occasion. L'ouvrage [8] est plus abordable, contient de nombreux problèmes amusants, et se focalise sur la façon dont une personne habituée aux raisonnements mathématiques peut appréhender l'algorithmique et la programmation. Les exemples sont donnés en pseudo-code Pascal. Enfin, [3] est une introduction assez courte à l'algorithmique destinée au premier cycle. Les propos sont très clairs mains moins approfondis que dans les deux autres ouvrages.
- Dans les ouvrages un peu plus techniques, mais néanmoins centrés sur l'algorithmique, on trouvera [7] ou [6] respectivement consacrés à l'écriture d'algorithmes en C et C++.
- En ce qui concerne particulièrement le C, [5] est une référence et présente la norme de 1983. Plus agréable à lire, et tout aussi complet, on pourra consulter [2] qui contient de nombreux exemples.
- Enfin, il existe un grand nombre d'ouvrages très techniques, comme [1] qui traite de programmation système (sous Unix, mais les principes sont valables sous Windows), [9], qui constitue une référence de l'Api d'OpenGL et de nombreux autres ouvrages...

Chapitre 1

Structures de données

Dans ce chapitre, nous présentons les structures de données classiques, dynamiques et basée pour leur implantation sur l'utilisation de pointeurs. Nous aborderons ainsi les listes, les piles, les arbres binaires et les graphes.

1 Notion de pointeur

Nous avons vu qu'une variable désignait en fait un «tiroir» dans la mémoire de l'ordinateur, dans lequel nous pouvions ranger une valeur. Nous pouvons considérer qu'un pointeur est aussi un tiroir, mais qu'il contient une référence à un autre tiroir. Pour préciser la forme que prend cette référence, nous pouvons continuer dans le même sens en précisant que tous les tiroirs dont dispose la machine sont numérotés. Le numéro d'un tiroir est appelé son adresse. Si la variable `a` correspond à tel tiroir, et que ce tiroir est le tiroir n°2400 de la machine, un pointeur qui fait référence à `a` sera un tiroir qui contiendra le numéro 2400.
Sur la figure 1.1 sont représentées deux variables. Une variable «ordinaire» : `b`, et un pointeur (qui est aussi une variable) : `p`.

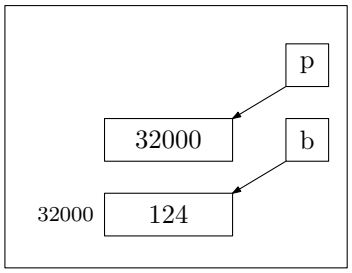


Fig. 1.1 – Une variable et un pointeur

Le pointeur `p` contient la valeur 32000, qui est l'adresse de la variable `b`. Dans une telle situation, on dit que `p` *pointe* sur `b`. Nous utiliserons des notations (un peu) similaires au C, comme :

`b`	pour désigner le contenu de (du tiroir) `b` (124)
`p`	pour désigner le contenu de (du tiroir) `p` (32000)
`&b`	pour désigner *l'adresse* du tiroir `b` (ici 32000)
`*p`	pour désigner *le contenu du tiroir sur lequel pointe* `p` (124)

2 Passage par adresse

La première utilisation des pointeurs que nous verrons est le passage des paramètres par adresse. Considérons les procédures données en 1.1.

ALG. 1.1 Passage d'un paramètre par valeur

```
ajoute_un(a : entier)
    a←a+1

algo_test()
    b : entier
    b←3
    ajoute_un(b)
    écrire(b)
```

La procédure algo_test affiche la valeur 3 (et non 4), comme illustré sur la figure 1.2. La variable b contient la valeur 3 (1.2-a), puis la procédure ajoute_un est appelée avec, comme paramètre, le contenu de b, c'est à dire la valeur 3. À l'intérieur de la procédure ajoute_un, (1.2-b) a est donc initialisé à la valeur 3. On ajoute 1 à la valeur de a (1.2-c), et a vaut donc 4. La procédure se termine. La valeur de b (1.2-d) est toujours 3.

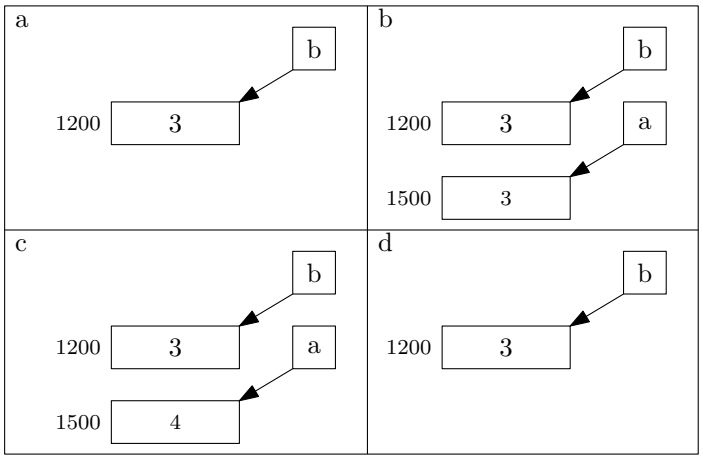


FIG. 1.2 – Exécution de algo_test

Dans certains cas, nous souhaitons que les paramètres soient effectivement modifiés lors d'un appel de procédure, même s'il faut se méfier de telles pratiques. On voit rapidement que, pour rendre ceci possible, une solution serait, plutôt que de communiquer la valeur qui est dans b, de communiquer *l'adresse* du tiroir b. Ceci permettra à la procédure (ou fonction) d'écrire réellement *dans* le tiroir b.

La nouvelle procédure principale est indiquée en 1.2.

C'est donc *l'adresse* de b qui est communiquée. Naturellement nous devons réécrire notre procédure ajoute_un en une procédure ajoute_un_bis qui reçoit maintenant en paramètre l'adresse d'une variable plutôt qu'un entier. L'adresse d'une variable étant stockée dans un pointeur, ajoute_un_bis aura donc un pointeur (p par exemple) pour paramètre.

Nous voulons ajouter 1 au *contenu* de la variable pointée par p, et non pas ajouter 1 au contenu de p. Il faudra donc écrire : *p←*p+1. Ce que nous obtenons est donné en 1.2.

En effet, si b vaut 3 (figure 1.3-a) et est stocké dans le tiroir qui a pour adresse 1200, p sera initialisé à 1200 (1.3-b) à l'appel de la procédure et *p vaudra le contenu du tiroir dont l'adresse est 1200, c'est à dire 3. *p+1 vaudra donc 4 (1.3-c) et cette valeur sera rangée dans le tiroir dont l'adresse est stockée dans p, c'est à dire dans le tiroir 1200. Au retour de la procédure (1.3-d) b vaudra donc 4.

Ce type de passage des paramètres, qui permet de modifier les variables de l'algorithme appelant se nomme passage par adresse, tout simplement parce que c'est l'adresse de la variable, plutôt que sa valeur qui est transmise au sous programme ou à la fonction.

Alg. 1.2 Passage d'un paramètre par adresse

```
ajoute_un_bis(p : pointeur)
    *p←*p+1

algo_test_bis()
    b : entier
    b←3
    ajoute_un_bis(&b)
    écrire(b)
```

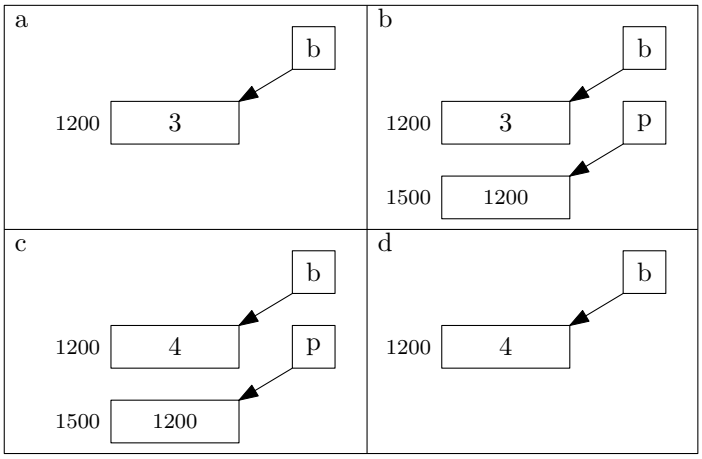


Fig. 1.3 – Exécution de algo_test_bis

3 Listes chaînées

On représente généralement une liste comme indiqué sur la figure 1.4.

La liste simplement chaînée du schéma 1.4 est une liste de 4 entiers. Le premier entier de la liste vaut 4, le second 12, etc... Comment représenter plus «informatiquement» une telle structure ? La solution réside dans l'utilisation d'enregistrements et de pointeurs. Afin de rendre les notations plus explicites dans la suite, nous allons ajouter un peu de vocabulaire à notre langage algorithmique et prendrons l'habitude, lorsque nous déclarons un pointeur, de préciser vers quel type de donnée il est censé pointer. Considérons la définition suivante :

Alg. 1.3 Définition d'une liste

```
enregistrement cellule
    val : entier
    suiv : pointeur cellule
```

Cette définition du type cellule correspond à l'une des quatre cases du schéma. Chaque case est en effet composée d'une valeur entière et d'un «chemin» vers la case suivante, un pointeur dans notre cas. La liste en elle-même est accessible par un simple pointeur vers une cellule (le carré du haut sur le schéma).

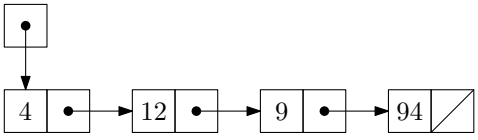


Fig. 1.4 – Une liste simplement chainée

3.1 Parcours de liste

Voici une première procédure algorithmique qui prend une liste en paramètre, et la parcourt en affichant ses valeurs. Elle utilise une valeur spéciale, que nous appellerons nul et qui est la valeur écrite dans un pointeur qui ne pointe sur rien (c'est souvent 0 en pratique) :

ALG. 1.4 Affichage d'une liste

```
affiche_liste(p : pointeur cellule)
    répéter tant que p≠ nul
        écrire ((*p).val)
        p←(*p).suiv
```

La ligne «écrire ((*p).val)» signifie : écrire le champ val de la cellule pointée par p, et la ligne «p←(*p).suiv» signifie : mettre dans p la valeur inscrite dans le champ suiv de la cellule pointée par p... La figure 1.5 résume l'exécution de l'algorithme sur une liste contenant les valeurs 1, 2 et 3. Pour en simplifier la lecture, une flèche en pointillés a été ajoutée pour visualiser vers quelle case de la mémoire pointe chaque pointeur.

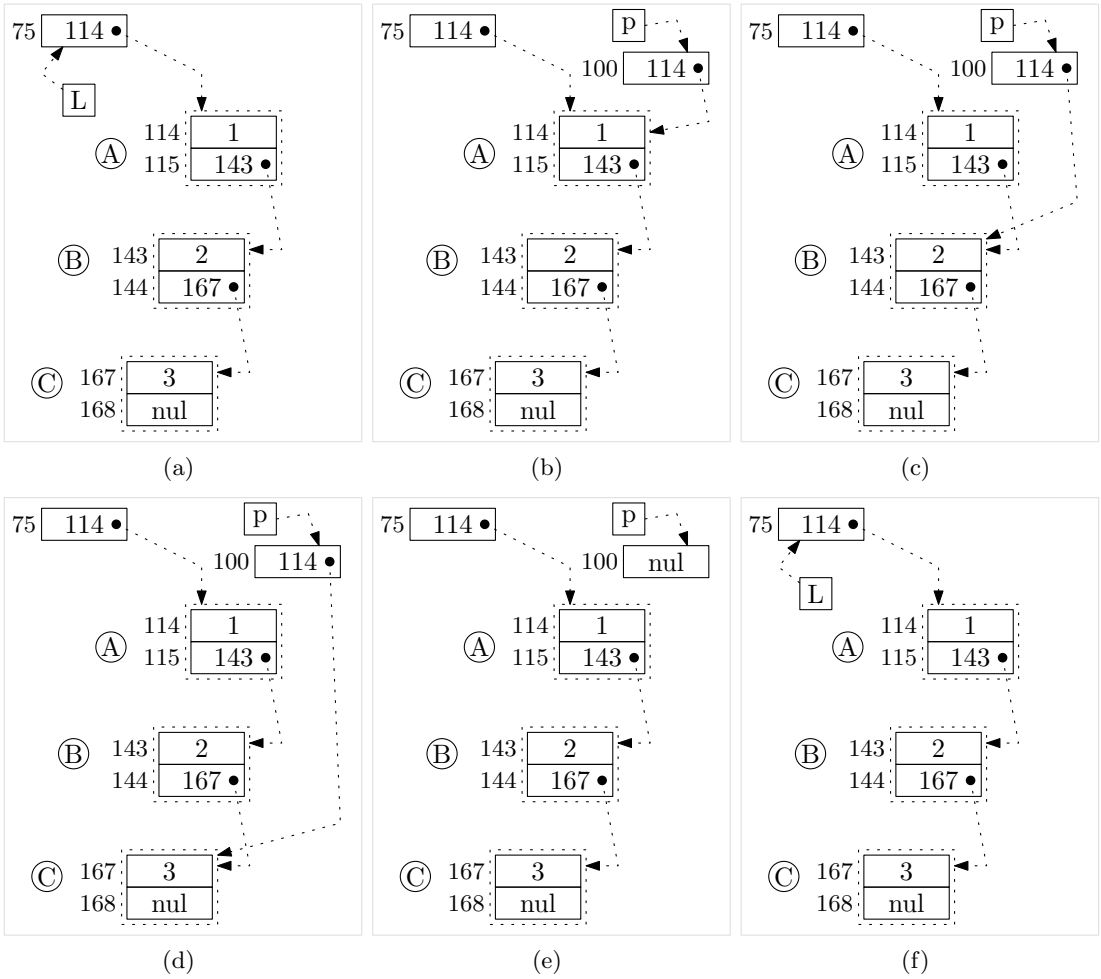


FIG. 1.5 – Affichage d'une liste

Supposons que la liste du schéma (a) ait été créée et que le pointeur L pointe sur la première cellule. Lorsqu'on appelle la procédure affiche_liste(L), le pointeur p est créé (b) et initialisé à la valeur contenue dans L (114). Puisque p ne vaut pas nul, la boucle est engagée. Le champ val de la cellule pointée par p (cellule A) est affiché (1 s'affiche). Puis, la valeur du champ suiv de la cellule pointée par p (cette valeur est 143) est inscrite dans p (c), qui pointe donc maintenant vers la cellule B. Puisque p ne vaut pas nul, la boucle continue. Le champ val de la cellule pointée

par p (cellule B) est affiché (2 s'affiche). Puis, la valeur du champ suiv de la cellule pointée par p (cette valeur est 167) est inscrite dans p (d), qui pointe donc maintenant vers la cellule C. Puisque p ne vaut pas nul, la boucle continue. Le champ val de la cellule pointée par p (cellule C) est affiché (3 s'affiche). Puis, la valeur du champ suiv de la cellule pointée par p (cette valeur est nul) est inscrite dans p, (e) qui ne pointe donc plus vers rien. Puisque p vaut nul, la boucle s'arrête. Lorsque la procédure est terminée, la liste a été affichée et est restée intacte (f) en mémoire.

3.2 Insertion en tête de liste

L'algorithme suivant insère un entier en tête de la liste passée en paramètres. Il utilise la notion d'allocation dynamique : la création d'un emplacement mémoire au cours de l'exécution.

Alg. 1.5 Insertion en tête de liste

```
insère_liste(p : pointeur cellule, i : entier) : pointeur cellule
    p1 : pointeur cellule
    p1←nouvelle cellule
    (*p1).suiv←p
    (*p1).val←i
    renvoyer p1
```

Supposons que la liste du schéma (a) (figure 1.6) ait été créée et que le pointeur L pointe sur sa première cellule. On souhaite insérer la valeur 5 en tête, par un appel à L←insère_liste(L,5). Les variables i p et p1 de la fonction d'insertion sont créées (b). Le pointeur p est initialisé à l'adresse transmise (114) et la variable i à la valeur à insérer (5). Lors de l'exécution de p1←**nouvelle** cellule, une nouvelle cellule vide est créée en mémoire (c), et son adresse est rangée dans le pointeur p1. Puis ((*p1).suiv←p), le champ suiv de la cellule pointée par p1 reçoit la valeur stockée dans p (d). Ensuite ((*p1).val←i), le champ val de la cellule pointée par p1 reçoit la valeur stockée dans i (e). Enfin, (renvoyer p1), la valeur contenue dans le pointeur p1 est renvoyée par la fonction (f). Puisque l'appel à cette dernière était : L←insère_liste(L,5), la valeur renvoyée (152) est stockée dans L (g). On a donc bien inséré la valeur 5 en tête de la liste L. Notons au passage que l'algorithme précédent fonctionne aussi si la liste est initialement vide, c'est à dire si L vaut nul.

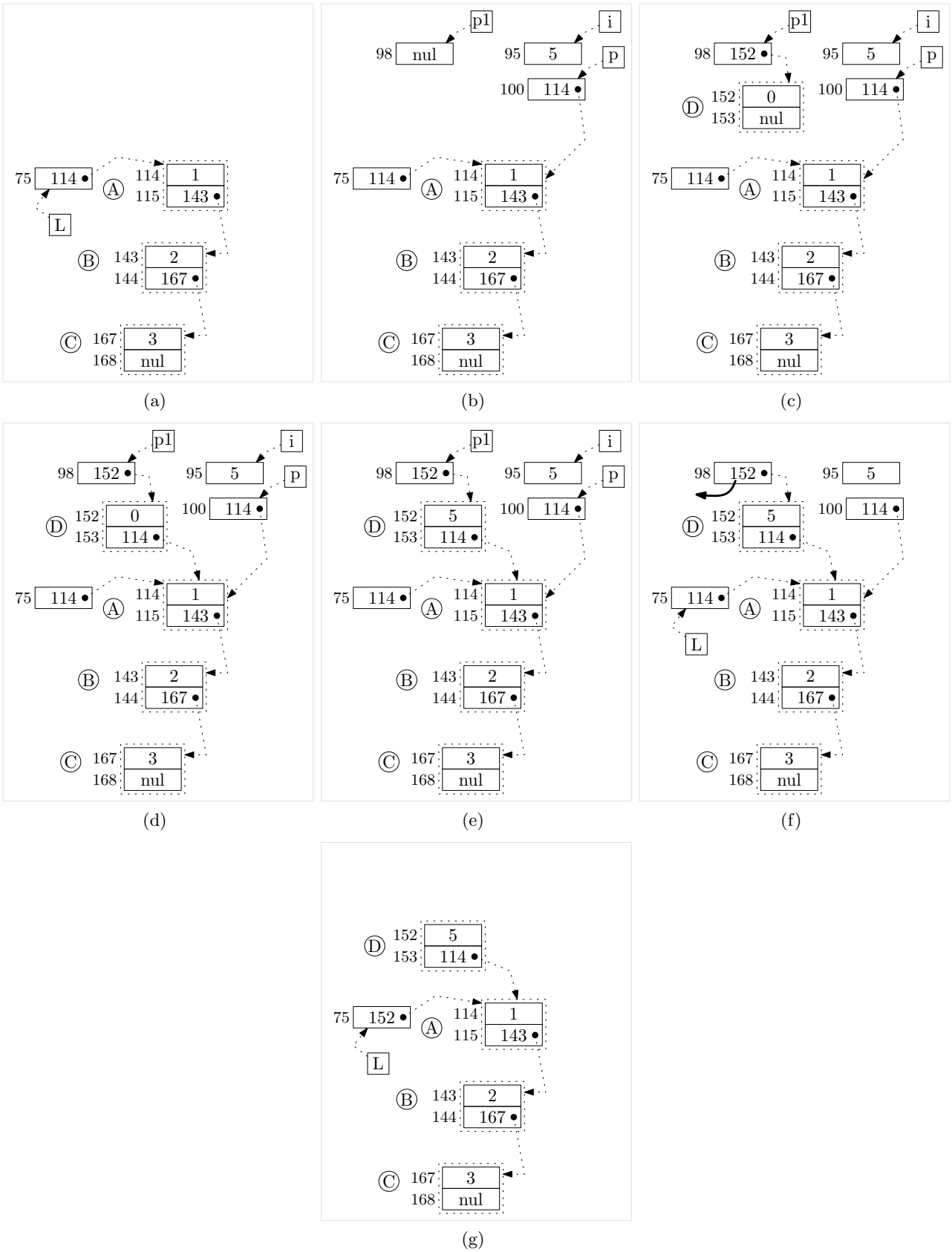


FIG. 1.6 – Insertion dans une liste

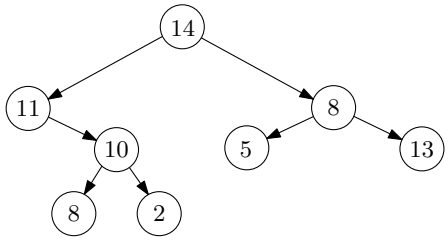


FIG. 1.7 – Un arbre binaire

4 Arbres binaires

Un arbre binaire est représenté figure 1.7. Chaque *nœud* de l'arbre contient une donnée (dans notre exemple il s'agit d'entiers) et possède 0, 1 ou 2 *fils*. Les *feuilles* de l'arbre sont les nœuds qui n'ont pas de fils (elles sont en bas), et la *racine* de l'arbre est le nœud qui n'a pas de *père* (elle est en haut !). On peut représenter informatiquement un arbre en utilisant enregistrements et pointeurs :

ALG. 1.6 Définition d'un arbre binaire

```
enregistrement nœud
    val : entier
    gauche : pointeur nœud
    droite : pointeur nœud
```

Chaque nœud possède ainsi une valeur (le champ val), ainsi que deux pointeurs vers les nœuds fils (gauche et droite). Si un fils est absent, on mettra simplement à nul le pointeur correspondant.

Notons que la structure d'un arbre est récursive. On peut définir un arbre ainsi : Un arbre peut être vide ou peut être un nœud qui a pour fils un ou deux arbres... En conséquence, les algorithmes sur les arbres sont souvent récursifs eux aussi.

4.1 Parcours d'un arbre

La procédure récursive suivante affiche chaque nœud d'un arbre :

ALG. 1.7 Affichage d'un arbre binaire

```
affiche_prefixe(r : pointeur nœud)
    si r≠nul alors
        écrire((*r).val)
        affiche_prefixe((*r).gauche)
        affiche_prefixe((*r).droite)
```

On obtient de cette façon un affichage préfixé, c'est à dire l'affichage de la valeur d'un nœud avant l'affichage de ses arbres fils. Dans le cas de la figure 1.7, on obtiendrait : 14 11 10 8 2 8 5 13

Il existe de même les algorithmes d'affichage infixé ou suffixé.

4.2 Construction d'un arbre

L'algorithme suivant permet de construire un arbre. Il prend en paramètres la valeur du futur nœud racine et des pointeurs vers les sous-arbres gauche et droite. Il renvoie un pointeur vers l'arbre construit. Chacun des pointeur vers les sous-arbres peut être «nul».

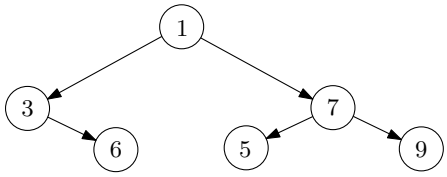


FIG. 1.8 – Un autre arbre binaire

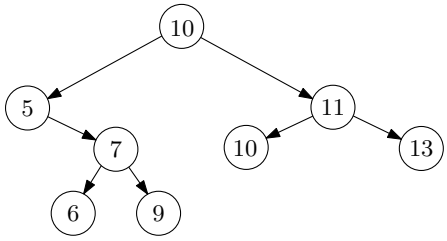


FIG. 1.9 – Un arbre binaire de recherche

ALG. 1.8 Construction d'un arbre binaire

```
construit(v : entier, ag : pointeur nœud, ad : pointeur nœud) : pointeur nœud
    p : pointeur nœud
    p←nouveau nœud
    (*p).val←v
    (*p).gauche←ag
    (*p).droite←ad
    renvoyer p
```

L'arbre de la figure 1.8 est construit par l'exécution de :

ALG. 1.9 Exemple de construction d'une arbre binaire

```
    p1,p2 : pointeurs nœuds
    p1←construit(5,nul,nul)
    p2←construit(9,nul,nul)
    p2←construit(7,p1,p2)
    p1←construit(6,nul,nul)
    p1←construit(3,nul,p1)
    p1←construit(1,p1,p2)
```

L'arbre final est pointé par p1.

4.3 Arbres binaires de recherche

Un arbre binaire de recherche est un arbre binaire qui vérifie les propriétés suivantes :

1. Tous les nœuds du sous-arbre gauche ont une valeur strictement inférieurs à celle de la racine.
2. Tous les nœuds du sous-arbre droit ont une valeur supérieure ou égale à celle de la racine.
3. Les sous-arbres droit et gauche sont des arbres binaires de recherche.

L'arbre de la figure 1.9 est un arbre binaire de recherche. Un arbre binaire de recherche est utilisé, comme son nom l'indique, pour rechercher la présence d'une valeur dans un ensemble de valeurs. La fonction suivante renvoie un pointeur sur le nœud cherché et nul si la valeur n'est pas trouvée.

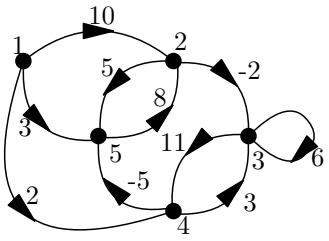


FIG. 1.10 – Un graphe orienté

ALG. 1.10 Recherche d'un élément dans un arbre binaire de recherche

```
recherche(a : pointeur nœud, v : entier) : pointeur nœud
    p : pointeur nœud
    si a≠nul alors
        si (*a).val=v alors
            renvoyer a
        sinon
            si (*a).val<v alors
                renvoyer recherche((*a).droite,v)
            sinon
                renvoyer recherche((*a).gauche,v)
    sinon
        renvoyer nul
```

Cet algorithme mérite quelques commentaires : D'une part, on préférera toujours renvoyer un pointeur vers le nœud que la valeur du nœud. En effet, connaissant le pointeur, il est facile d'avoir la valeur, alors que l'inverse n'est pas vrai. De plus le pointeur donne des indications sur la position du nœud dans l'arbre. En ce qui concerne l'efficacité de cet algorithme, on constate sans mal que la valeur à rechercher ne sera comparée à une autre valeur qu'un nombre de fois au pire égal à la *profondeur* de l'arbre (la longueur de la plus longue *branche*). Si l'arbre est parfait[1], cette hauteur sera de l'ordre de $\ln_2(n)$ (avec n le nombre de nœuds de l'arbre). En conséquence, la recherche d'une valeur prendra au pire $\ln_2(n)$ comparaisons, alors que dans un tableau non trié, elle prendrait au pire n comparaisons[2]. Les considérations de cet ordre (comptage de l'ordre de grandeur du nombre d'opérations en fonction de la taille des données) font partie de l'étude de la complexité des algorithmes et seront évoquées un peu plus loin.

5 Graphes

Un graphe orienté valué dans $\mathbb{R}$ est un ensemble N de sommets, et une application A de $N \times N$ dans $\mathbb{R} \cup \{nul\}$ qui à un couple de sommets fait correspondre la valeur de leur arête orientée ou l'absence d'arête. Si un graphe n'est pas orienté, alors $\forall (n_1, n_2) \in N^2, A(n_1, n_2) = A(n_2, n_1)$. Si un graphe n'est pas valué, l'application A est à valeurs dans $\{vrai, faux\}$ de façon à seulement signifier la présence ou l'absence d'arête. La figure 1.10 représente un exemple de graphe orienté.

Les sommets sont numérotés de 1 à 5, et nous avons : A(1,2)=10, A(2,1)=nul, A(4,3)=11, A(3,4)=3,...

Il existe de nombreuses manières de représenter un graphe dont : les matrices d'adjacence, les listes d'adjacences, les représentations par pointeurs.

[1] Un arbre parfait est tel que tous les niveaux, sauf éventuellement le dernier, sont remplis. Au dernier niveau, les feuilles sont groupées à gauche.

[2] Cette considération est importante. Si au lieu de 15 valeurs, l'arbre en contient 1 milliard, le nombre de comparaisons pour une recherche passera de 4 à seulement 30.

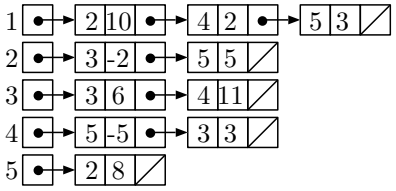


FIG. 1.11 – Représentation d'un graphe

5.1 Matrices d'adjacences

Soit un graphe (N, A) valué orienté à n sommets numérotés de 1 à n. La matrice d'adjacence de ce graphe est un tableau bidimensionnel t de taille $n \times n$ défini par : $\forall (i,j) \in \{1,...,n\}^2, \mathsf{t[i][j]} = A(i,j)$. Ainsi, le graphe de la figure 1.10 sera représenté par le tableau t suivant :

i\j	1	2	3	4	5
1	nul	10	nul	2	3
2	nul	nul	-2	nul	5
3	nul	nul	6	11	nul
4	nul	nul	3	nul	-5
5	nul	8	nul	nul	nul

5.2 Listes d'adjacences

On peut représenter un graphe par un tableau de listes (!), chaque élément de la liste étant un couple (n° de sommet, valeur d'arête). Le graphe de la figure 1.10 se représenterait comme indiqué figure 1.11.
Algorithmiquement, nous représenterions un graphe par une liste d'adjacence ainsi :

ALG. 1.11 Représentation d'une liste d'adjacence

```
enregistrement cellule
    sommet : entier
    valeur : réel
    suiv : pointeur cellule

t[1..n] : tableau de pointeurs cellules
```

5.3 Insertion d'une arête

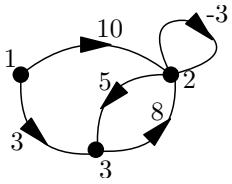
L'algorithme suivant insère une arête dans un graphe (les tableaux sont passés par adresse) :

ALG. 1.12 Insertion d'une arête dans un graphe

```
insertion(t[] : tableau de pointeurs cellules, s1 : entier, s2 : entier, v : réel)
p : pointeur cellule
    p ←nouvelle cellule
    (*p).sommet←s2
    (*p).valeur←v
    (*p).suivant←t[s1]
    t[s1]←p
```

5.4 Recherche de plus courts chemins

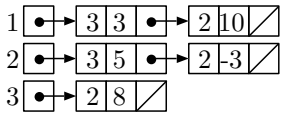
Nous nous intéressons à présent à la possibilité de rechercher le plus court chemin (le chemin qui emprunte le moins d'arêtes) entre deux sommets d'un graphe non orienté et non valué.



(a) Exemple de graphe

```
t[1..3] : tableau de pointeurs cellules
insertion(t,1,2,10)
insertion(t,1,3,3)
insertion(t,2,2,-3)
insertion(t,2,3,5)
insertion(t,3,2,8)
```

(b) Construction du graphe



(c) Liste d'adjacence associée

FIG. 1.12 – Construction et représentation d'un graphe

L'idée de l'algorithme est la suivante : Partant du sommet source, les distances aux autres sommets sont initialisées à $+\infty$. Puis, on sélectionne tous les voisins du sommet source, et on indique que leur distance au sommet source vaut 1. Ces sommets sont placés, avec le sommet source dans l'ensemble des sommets traités. Puis, on sélectionne l'ensemble des sommets voisins des derniers sommets traités, qui ne sont pas eux-même des sommets traités. On indique que leur distance au sommet source vaut 2. On les place dans la liste des sommets traités... jusqu'à ce que tous les sommets soient traités.
Nous allons donner un algorithme plus formel qui résout le problème du calcul des plus courts chemins. Cet algorithme prend en paramètres la matrice d'adjacence M du graphe, le nombre de sommets N, et le numéro du sommet source s. Les variables utilisées sont d[] qui contient les distances vers les autres sommets, ok[] qui comptabilise les sommets traités, encours[] qui indique les sommets que l'on traite, et marque[] qui indique quels sont les voisins des sommets que l'on traite qui n'ont pas déjà été traités. Les variables faits et dist indiquent respectivement le nombre de sommets déjà traités, et la distance en cours. Par souci de concision, le raccourci marque[]←0 utilisé dans l'algorithme signifie : mettre 0 dans toutes les cases du tableau marque[] et encours[]←marque[] est mis pour la recopie de toutes les valeurs du tableau marque[] dans le tableau encours[]. La figure 1.13 illustre le déroulement de l'algorithme pour un graphe de taille 7, de matrice d'adjacence (1 signifie la présence d'une arête et 0 l'absence d'une arête) :

i\j	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	1	0	0	1	1	1	0
3	1	0	0	1	0	0	0
4	0	1	1	0	1	0	0
5	0	1	0	1	0	0	1
6	0	1	0	0	0	0	1
7	0	0	0	0	1	1	0

Les plus courts chemins sont calculés à partir du sommet 1. Les sommets qui sont marqués dans ok[] sont en gras, ceux qui sont marqués dans encours[] sont en zone pointillée, et ceux qui sont marqués dans marque[] en zone hachurée. Près de chaque sommet figure sa valeur indiquée dans le tableau d[]. Une absence de valeur signifie $+\infty$. La figure 1.13 indique l'état du graphe, avant l'exécution de l'algorithme (a), après la ligne faits←1 (b), puis à deux endroits de la boucle principale : avant la ligne encours[]←marque[] ((c), (e) et (g)), et après la ligne marque[]←0 ((d), (f) et (h)).

ALG. 1.13 Recherche dun plus court chemin dans un graphe

```
plus_court(N : entier, M[1..N][1..N], s : entier) : tableau d'entiers
    d[1..N], ok[1..N], encours[1..N], marque[1..N] : tableaux d'entiers
    i, j, dist, faits : entiers
    d[]←+∞
    ok[]←0
    encours[]←0
    marque[]←0
    dist←0
    d[s]←dist
    ok[s]←1
    encours[s]←1
    faits←1
    répéter tant que faits≠N
        dist←dist+1
        répéter pour i de 1 à N
            si encours[i]=1 alors
                répéter pour j de 1 à N
                    si M[i][j]=1 et ok[j]≠1 alors
                        marque[j]←1
                        d[j]←dist
                        faits←faits+1

        encours[]←marque[]
        répéter pour i de 1 à N
            ok[i]←ok[i]+marque[i]

        marque[]←0

    renvoyer d
```

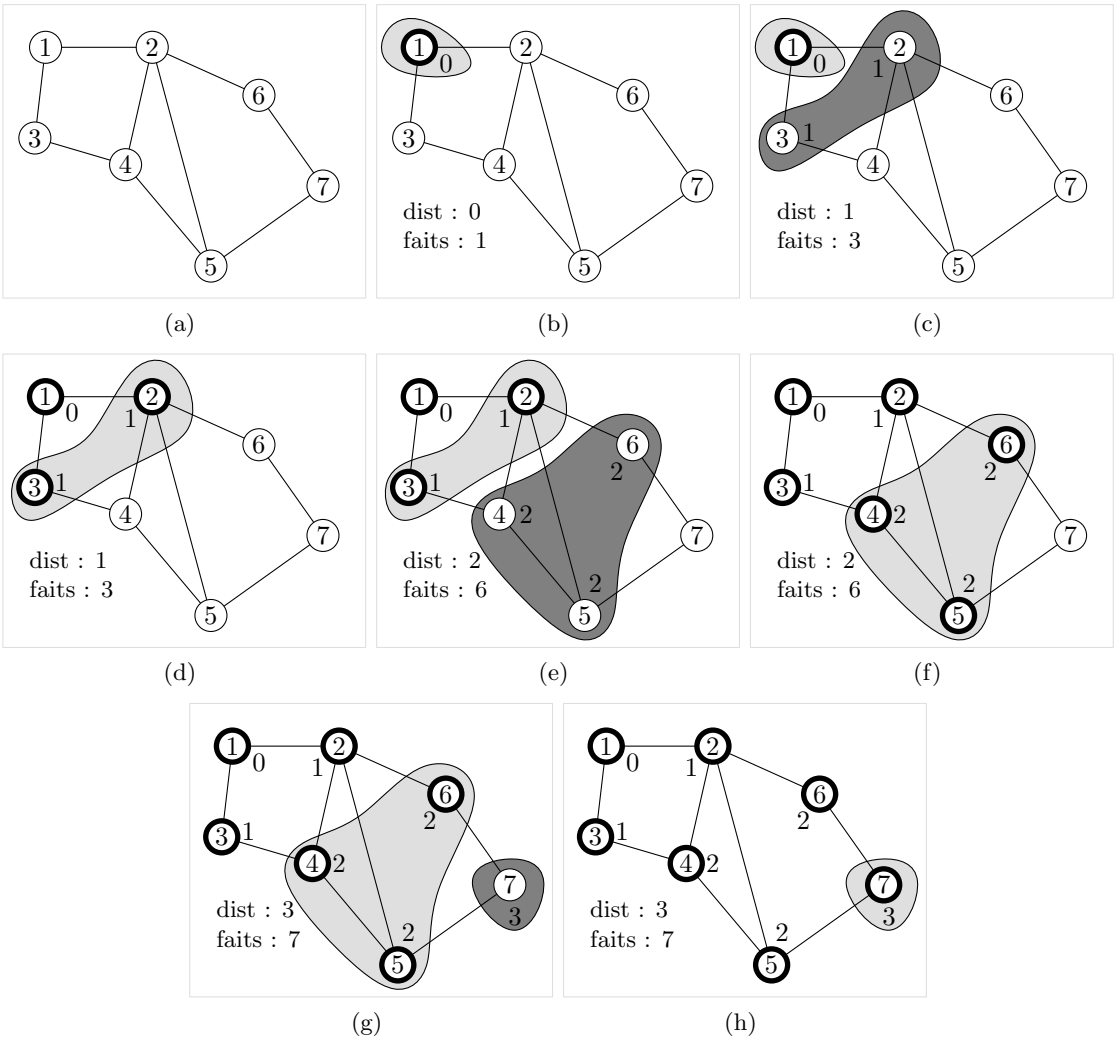


FIG. 1.13 – Recherche du plus court chemin

Chapitre 2

Méthodes algorithmiques

Un certain nombre de principes algorithmiques fondamentaux sont récurrents dans la résolution de nombreux problèmes. C'est le cas du principe très général de récursivité qui a été abordé l'an passé. Dans ce chapitre, nous verrons le principe du *divide and conquer* et de la programmation dynamique. Puis, nous aborderons le très vaste problème de la mesure de la complexité des algorithmes en prenant pour exemples quelques algorithmes de tri.

1 Diviser pour régner

La méthode «diviser pour régner» est aussi connu sous les noms «diviser pour résoudre» ou «divide and conquer». C'est une méthode récursive.
L'idée générale consiste à diviser la tâche à accomplir en deux sous-tâches deux fois plus simples, à résoudre ces sous-tâches, puis à rassembler les résultats des deux sous-tâches pour obtenir le résultat de la tâche complète. La résolution des deux sous-tâches se fait naturellement aussi par la même méthode (division en sous-sous-tâches). La méthode étant récursive, il faut nécessairement qu'on puisse résoudre *directement* une tâche très simple (condition d'arrêt de la récursivité)

1.1 Marquage d'une règle

Cet exemple est tiré de [7] et illustre bien la méthode du diviser pour régner (ainsi que son pendant itératif).
Il s'agit de tracer des graduations sur une règle, de longueurs différentes, comme indiqué sur la figure 2.1.
Sur cette figure, les nombres indiquent l'ordre dans lequel les graduations ont été tracées.
Le tracé prédédent est obtenu par l'algorithme 2.1, en exécutant : `regle(0,8,4)`.

Alg. 2.1 Marquage de repères sur une règle, version diviser pour régner

```
regle(g,d,h : entiers)
    m : entier
    m←(g+d)/2
    si  h>0 alors
        regle(g,m,h-1)
        regle(m,d,h-1)
        marque en m de hauteur h
```

Cet algorithme suit le modèle du diviser pour régner ; la graduation de la règle se fait en trois temps :

Fig. 2.1 – Règle tracée par l'algorithme récursif 2.1

FIG. 2.2 – Règle tracée par l'algorithme itératif 2.2

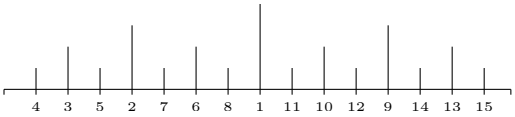


FIG. 2.3 – Comment est tracée cette règle ?

1. Graduer la partie gauche de la règle.
2. Graduer la partie droite de la règle.
3. «Réunir» les graduations en ajoutant la graduation centrale.

Néanmoins, il existe dans ce cas précis un algorithme itératif simple à trouver, ce qui n'est pas toujours le cas. En effet, dans l'exemple de la règle, le traitement qui divise les données, et celui qui les réunit sont relativement simples.
À titre d'exemple, nous donnons l'algorithme 2.2 qui trace la règle de manière itérative. L'ordre du tracé est visible sur la figure 2.2.

ALG. 2.2 Marquage de repères sur une règle, version itérative

```
regle_iter(g,d,h : entiers)
    lepas,ht,i : entiers
    lepas←1
    répéter pour ht variant de 1 à h
        répéter pour i variand de g+lepas/2 à d par pas de lepas
            marque en i de hauteur ht

        lepas←lepas*2
```

Observez bien l'ordre dans lequel se fait le tracé. Pensez-vous pouvoir produire un algorithme qui trace les marques comme indiqué dans la figure 2.3 ?

1.2 Tri par fusion

Le tri par fusion utilise la méthode du diviser pour régner. Le tri d'un tableau d'entiers se fait comme indiqué informellement par l'algorithme 2.3.

ALG. 2.3 Méthode générale du tri par fusion

```
pour trier le tableau t[ ]
    diviser t en deux tableaux t1 et t2
    trier t1
    trier t2
    fusionner les deux tableaux t1 et t2 pour obtenir le résultat
```

2 Programmation dynamique

Alors que le principe du «diviser pour régner» consiste à découper un gros problème en deux (parfois plus) sous problèmes plus simple, la programmation dynamique consiste à résoudre tous les sous-problèmes d'un problème

donné, afin d'utiliser les résultats des sous-problèmes pour résoudre le problème final. Cette méthode est à appliquer lorsque :
– la solution s'obtient récursivement ;
– un algorithme récursif ordinaire implique le calcul d'un grand nombre de sous-problèmes, alors que peu d'entre eux sont différents ;
– ...
Nous allons éclaircir ces notions sur un exemple classique : l'ordre de multiplication d'une chaîne de matrices.
Considérons les 3 matrices, de tailles suivantes :

$$\begin{array}{ll} M_0 & (2,4) \\ M_1 & (4,3) \\ M_2 & (3,5) \end{array}$$

Pour effectuer le produit $M_0M_1M_2$, nous avons le choix de l'ordre des multiplications. Nous pouvons effectuer $M_0(M_1M_2)$ ou bien $(M_0M_1)M_2$.
Le produit de deux matrices de tailles respectives (p,r) et (r,q) est une matrice de taille (p,q). Chaque coefficient du résultat est une somme de r termes. Il y a donc environ pqr opérations à effectuer pour calculer ce produit.
Le tableau suivant indique, selon l'ordre du produit choisi, le nombre d'opérations nécessaires :

$$\begin{array}{ll} M_0(M_1M_2) & 4\times 3\times 5 + 2\times 4\times 5 = 100 \\ (M_0M_1)M_2 & 2\times 4\times 3 + 2\times 3\times 5 = 54 \end{array}$$

On voit donc que l'ordre des opérations a une certaine importance.
Déterminer l'ordre optimal de multiplication d'une chaîne de matrices peut être résolu par programmation dynamique.
La solution s'obtient récursivement : notons l_i le nombre de lignes de la matrice i, c_i le nombre de colonnes de la matrice i et $\sigma_{i,i+j}$ le nombre minimal d'opérations pour calculer le produit : $M_i..M_{i+j}$. Nous pouvons écrire que : $(M_i...M_{i+j}) = (M_i...M_{i+k})(M_{i+k+1}..M_{i+j})$ et que cet ordre particulier implique un nombre minimal de $\sigma_{i,i+k} + \sigma_{i+k+1,i+j} + l_ic_{i+j}c_{i+k}$ opérations. Si on recherche la valeur de k comprise entre 0 et j qui minimise cette valeur, on obtient $\sigma_{i,i+j}$.
Dans ce cas précis de récursivité, on a intérêt à utiliser la programmation dynamique, plutôt qu'une écriture récursive immédiate. Dans le dernier cas, on obtiendrait pour la multiplication de 5 matrices, l'arbre des appels représenté sur la figure 2.4.

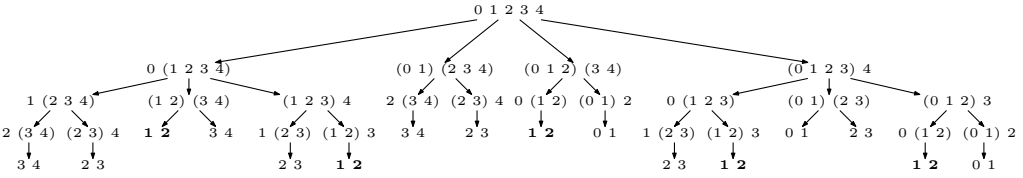


FIG. 2.4 – Arbre des appels – Multiplication d'une chaîne de matrices – version récursive

Cette figure se lit ainsi : pour calculer le nombre d'opérations optimal, du produit $M_1(M_2M_3M_4)$ (troisième nœud en lecture préfixée), il faut calculer les nombres d'opérations optimaux pour les produite $M_2(M_3M_4)$ et $(M_2M3)M_4$ et pour faire ceci il faudra calculer le nombre d'opérations pour les produits M_3M_4 et M_2M_3. Certaines feuilles de l'arbre sont en gras. Elles nous permettent de remarquer que pour avoir la réponse à notre question de départ, il faudra calculer 5 fois le nombre d'opérations nécessaires au produit M_1M_2.
L'idée de la programmation dynamique est de ne pas provoquer plusieurs évaluations du même sous-problème. Pour cela, on décide de calculer *en premier* les plus petits sous-problèmes, desquels on déduit les solutions au sous-problèmes de taille supérieure, desquels on déduit... L'ordre des calculs tels que nous venons de le décrire est indiqué sur la figure 2.5.

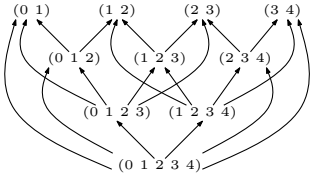


FIG. 2.5 – Ordre des calculs – Multiplication d'une chaîne de matrices – programmation dynamique

Si nous faisons les calculs dans l'ordre indiqué par la figure 2.5, nous allons d'abord calculer $\sigma_{0,1}$, $\sigma_{1,2}$, $\sigma_{2,3}$ et $\sigma_{3,4}$ et *stocker* les résultats. Puis, nous allons calculer $\sigma_{0,2}$, $\sigma_{1,3}$ et $\sigma_{2,4}$ en utilisant les résultats stockés et stocker ces nouveaux résultats. Par la suite, on voit par exemple que le calcul de $\sigma_{0,3}$ utilisera les résultats précédemment stockés : $\sigma_{0,1}$, σ_{02}, σ_{13} et σ_{23}.
Il nous faut donc une structure de stockage pour les résultats intermédiaires, ce qui ne serait pas nécessaire dans la version récursive. Nous utiliserons le tableau bidimensionnel cout[ ][ ] (cout[i][j] contiendra le nombre optimal d'opérations pour effectuer le produit $M_i..M_j$. Un problème classique de programmation dynamque est de «tracer» la façon dont le résultat est obtenu. En effet, si nous nous contentons de calculer le tableau cout, nous obtiendrons, dans le cas des trois matrices précédentes :

0	24	54
	0	60
		0

Ce tableau se lit ainsi : le nombre optimal d'opérations pour calculer le produit M_1M_2 est indiqué ligne 1, colonne 2 (60). Le nombre minimal d'opérations pour calculer le produit $M_0..M_2$ est indiqué ligne 0, colonne 2 (54). Nous avons donc la réponse à la question : on peut calculer le produit $M_0..M_2$ en seulement 54 opérations. Mais nous ne savons pas comment... (dans ce cas précis, c'est évident, puisque $\sigma_{1,2} = 60 > 54$, mais si le tableau est plus grand...)
Nous devons donc, en plus de calculer la solution, mettre en place un moyen de retrouver la façon dont la solution a été construite (54 vient-il de $M_0(M_1M_2)$ ou de $(M_0M_1)M_2$?)
Penchons-nous tout d'abord sur le calcul du tableau cout. Une première version est donnée par l'algorithme 2.4.

ALG. 2.4 Problème de la multiplication d'une chaîne de matrices (version provisoire)

```
chaine(taille[][] : tableau d'entiers,n : entier) : tableau bidimensionnel d'entiers
    cout[][] : tableau d'entiers de dimension n,n
    i,j,k,nb : entiers
    {Initialisation}
    répéter pour i de 0 à n-1
        répéter pour j de 0 à n-1
            cout[i][j]←∞

        cout[i][i]←0


    {Calcul de σ}
    répéter pour j de 1 à n-1
        répéter pour i de 0 à n-1-j
            répéter pour k de 0 à j-1
                nb←cout[i][i+k]+cout[i+k+1][i+j]+taille[i][0]*taille[i+j][1]*taille[i+k][1]
                si nb<cout[i][i+j] alors
                    cout[i][i+j]←nb


    renvoyer cout
```

Le paramètre taille est un tableau à n lignes et 2 colonnes. n est le nombre de matrices, taille[i][0] indique le nombre de lignes de M_i et taille[i][1] son nombre de colonnes. Nous ne faisons pas ici de vérification sur les tailles.
La première partie de l'algorithme initialise le tableau cout : les produits $\sigma_{i,i}$ sont initialisés à 0 et toutes les autres valeurs sont mises à $+\infty$.
L'indice j de la boucle la plus extérieure est la taille du sous-problème considéré. On commence avec des produits de 2 matrice (M_iM_{i+1}) jusqu'à des produits de n matrices $(M_0..M_{n-1})$. Un tour de boucle sur j va donc calculer tous les $\sigma_{i,i+j}$ (et le résultat qui nous intéresse est $\sigma_{0,n-1}$.

À l'intérieur de cette boucle, la boucle sur i fait varier la première matrice du produit. À j fixé, on va donc calculer : $\sigma_{0,j}$, $\sigma_{1,1+j}$,... $\sigma_{n-1-j,n-1}$. À l'intérieur de la boucle sur i nous devons donc calculer $\sigma_{i,i+j}$.
Pour cela, nous utilisons la relation de récurrence précédente :

$$\sigma_{i,i+j} = \min_{k\in[0..j]} (\sigma_{i,i+k} + \sigma_{i+k+1,i+j} + l_i c_{i+j} c_{i+k})$$

Le calcul de ce minimum est réalisé par la boucle sur k.
Enfin, nous renvoyons le résultat qui est dans le tableau cout.
Nous voyons dans l'algorithme 2.4 qu'une coupure particulière du produit $M_i..M_{i+j}$ en $(M_i..M_{i+k})(M_{i+k+1}..M_{i+j})$ est utilisée dans la solution optimale pour la valeur de k qui correspond au minimum. Pour reconstruire la solution (l'ordre de multiplication des matrices), c'est la valeur de $i+k$ du point de coupure dont nous avons besoin. Nous allons donc ajouter à notre solution un second tableau bidimensionnel, nommé ordre[ ][ ] pour stocker le point de coupure des produits : si ordre[i][i+j] vaut i+k, cela signifiera que pour calculer de façon optimal le produit $M_i..M_{i+j}$, il faut calculer : $(M_i..M_{i+k})(M_{i+k+1,i+j})$. La version finale est donnée par l'algorithme 2.5. De plus, comme nous devons renvoyer deux tableaux, nous prenons une écriture proche du C : les tableaux cout et ordre sont passés en paramètres et par adresse.

Alg. 2.5 Problème de la multiplication d'une chaîne de matrices (version finale)

```
chaine(taille[][], ordre[][], cout[][] : tableaux d'entiers,n : entier)
    i,j,k : entiers
    {Initialisation}
    répéter pour i de 0 à n-1
        répéter pour j de 0 à n-1
            cout[i][j]←∞

        cout[i][i]←0


    {Calcul de σ}
    répéter pour j de 1 à n-1
        répéter pour i de 0 à n-1-j
            répéter pour k de 0 à j-1
                nb←cout[i][i+k]+cout[i+k+1][i+j]+taille[i][0]*taille[i+j][1]*taille[i+k][1]
                si nb<cout[i][i+j] alors
                    cout[i][i+j]←nb
                    ordre[i][i+j]←i+k
```

3 Notions de complexité en temps

Les différents problèmes que nous sommes amenés à traiter peuvent être résolus de différentes manières (selon l'algorithme choisi). Un des critères pour choisir un algorithme plutôt qu'un autre est le temps que mettra celui-ci à résoudre le problème. L'étude du temps d'exécution des algorithmes[1] constitue une partie d'un domaine vaste nommé *complexité*. Dans cette section, nous nous attacherons à découvrir comment calculer la complexité en temps de certains algorithmes.
Le temps d'exécution en secondes d'un *programme* dépend :
– des entrées qui lui sont fournies ;
– de la machine qui l'exécute.

[1] Il s'agit bien du temps d'exécution des algorithmes, et non des programmes sur une machine, le second étant forcément relié au premier, et le premier étant une caractéristique intrinsèque de l'algorithme choisi, indépendante des avancées technologiques.

Le même programme, exécuté sur la même machine, auquel on fourni les même entrées, mettra à chaque fois le même temps à s'exécuter (ce qui n'est pas évident à l'époque des ordinateurs de bureau multitâches).
Si on considère une machine B deux fois plus rapide qu'une machine A (c'est à dire dont les instructions élémentaires sont deux fois plus rapides), alors le programme mettra environ deux fois moins de temps à s'exécuter. En revanche, si sur la même machine A, on exécute deux fois un programme, la première fois avec N données d'entrées, et la seconde fois avec $2N$ données d'entrées, il se peut que le programme mette 2 fois plus de temps à s'exécuter, ou bien 4 fois plus de temps ou autre chose encore.
Autrement dit, ce qui est *vraiment* important, c'est de décrire ce que vaut le temps d'exécution en fonction de la taille des données, en faisant abstraction de la vitesse de la machine elle-même.

3.1 Tri par insertion

Voyons plus précisément de quoi il s'agit sur l'exemple du tri par insertion, rappelé dans l'algorithme 2.6.

ALG. 2.6 Tri par insertion

```
insertion(t[] : tableau d'entiers, n : entier)
    i,j : entiers
    val : entier
    répéter pour j variant de 1 à n-1
        val←t[j]
        i←j-1
        répéter tant que i≥0 et t[i]>val
            t[i+1]←t[i]
            i←i-1
        t[i+1]←val
```

Tableau	clé	
$4-6-5-2-7-3$		Début de l'algorithme
$4-⑥-5-2-7-3$	6	Sélection de la première clé
$\overset{\downarrow}{4}-6-5-2-7-3$	6	Test d'insertion
$4-⑥-5-2-7-3$	6	Insertion de la clé
$4-6-⑤-2-7-3$	5	Sélection de la seconde clé
$4-\overset{\downarrow}{6}-5-2-7-3$	5	Test d'insertion
$4-\overset{\downarrow}{6}-6-2-7-3$	5	Décalage
$\overset{\downarrow}{4}-6-6-2-7-3$	5	Test d'insertion
$4-⑤-6-2-7-3$	5	Insertion de la clé
...	...	...
$2-4-5-6-7-③$	3	Sélection de la dernière clé
$2-4-5-6-\overset{\downarrow}{7}-3$	3	Test d'insertion
$2-4-5-6-\overset{\downarrow}{7}-7$	3	Décalage
$2-4-5-\overset{\downarrow}{6}-7-7$	3	Test d'insertion
$2-4-5-\overset{\downarrow}{6}-6-7$	3	Décalage
$2-4-\overset{\downarrow}{5}-6-6-7$	3	Test d'insertion
$2-4-\overset{\downarrow}{5}-5-6-7$	3	Décalage
$2-\overset{\downarrow}{4}-5-5-6-7$	3	Test d'insertion
$2-\overset{\downarrow}{4}-4-5-6-7$	3	Décalage
$\overset{\downarrow}{2}-4-4-5-6-7$	3	Test d'insertion
$\overset{\downarrow}{2}-2-4-5-6-7$	3	Décalage
$2-③-4-5-6-7$	3	Insertion de la clé

Le principe de l'algorithme est simple et ressemble à la méthode que nous utilisons pour trier une poignée de cartes à jouer. Le tableau est parcouru de gauche à droite (boucle extérieure). Pour chaque valeur, la clé, on cherche à l'insérer à sa gauche (qui est supposée triée). Pour cela, on parcourt le tableau de droite à gauche, à partir de la case située juste à gauche de la clé (boucle intérieure). Dès qu'on a trouvé la position d'insertion, on insère la clé.
Nous allons à présent essayer de compter le nombre d'instructions élémentaires (le temps d'exécution de l'algorihtme) nécessaire à l'exécution du tri pour un tableau de taille n.
La boucle extérieure sera exécutée $n-1$ fois. Elle contient des instructions élémentaires, et une autre boucle. Nous devons nous intéresser aux nombre de tours effectués par la boucle intérieure. Un exemple d'exécution est détaillé à côté de l'algorithme.
Il apparaît clairement que le nombre de fois que la boucle intérieure s'exécute ne dépend pas seulement de n ou de la valeur de j : elle dépend aussi du *contenu* du tableau. Nous sommes donc amenés à considérer au moins deux calculs : celui de la complexité moyenne (sur toutes les données possibles, ou sur toutes les données probables), et celui de la complexité dans le cas le pire. Le deuxième cas est souvent plus facile à calculer, et il donne un majorant.

Essayons de déterminer le cas le pire. Lorsque la clé examinée est en position j, alors le pire des cas se produit si la clé doit être insérée *tout au début* du tableau. Dans ce cas, la boucle intérieure sera exécutée j fois.
Pour récapituler, notre algorithme est composé d'une boucle principale, exécutée $n-1$ fois, pour j variant de 1 à $n-1$. Cette boucle contient 3 instructions élémentaires, et une boucle, qui est exécutée j fois dans le cas le pire. Cette boucle intérieure contient elle même 2 instructions élémentaires. En partant de l'approximation qu'une instruction élémentaire a un temps d'exécution de 1, le temps d'exécution dans le cas le pire de notre algorithme de tri d'un tableau de taille n est donc :

$$T(n) = \sum_{j=1}^{j=n-1} (3+2j) = 3(n-1) + n(n-1) = n^2 + 2n - 3$$

Obtenir un résultat aussi précis n'est pas notre but, d'autant plus que toutes les instructions élémentaires n'ont pas réellement le même temps d'exécution et que nous n'avons pas compté les tests des boucles ni l'incrémentation des compteurs...
Ce qui nous intéresse est de savoir comment se comporte le temps d'exécution lorsque n est grand. Notre but n'est pas de décrire ici les subtilités des calculs d'asymptotes, mais nous rappelons néanmoins deux définitions :
– Pour une fontion $g(n)$ donnée, on note :

$$\Theta(g(n)) = \left\{ f(n), \exists (c_1, c_2, n_0) \in R^2 \times N / \forall n > n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \right\}$$

– Pour une fontion $g(n)$ donnée, on note :

$$O(g(n)) = \{ f(n), \exists (c, n_0) \in R \times N / \forall n > n_0, 0 \leq f(n) \leq c g(n) \}$$

Dans notre exemple, $T(n) \in \Theta(n^2)$, ce qui est le résultat qui nous intéresse : dans le pire des cas, pour un grand nombre de données, le temps d'exécution se comportera comme le carré de la taille des données.
En pratique, on rencontre néanmoins très souvent l'autre notation, qui ne donne qu'une borne supérieure du comportement asymptotique. Puisque'on essaie de faire en sorte que cette borne soit la meilleure possible, la notation $O(.)$ peut la plupart du temps être comprise comme $\Theta(.)$.
Nous écrirons donc que l'algorithme du tri par insertion a un complexité dans le cas le pire en $O(n^2)$.
Intéressons nous maintenant au cas moyen. Nous allons supposer pour cela que les nombres du tableau ont été pris au hasard. Le point à éclaircir est le nombre de tours de la boucle intérieure, autrement dit, la position où la clé j devra être placée dans la portion triée $[0..j-1]$. Si les nombres ont été tirés au hasard, alors la portion triée contiendra en moyenne autant de nombre plus petits que la clé que de nombres plus grands que la clé. Et la position d'insertion sera donc $j/2$.
En reprenant notre calcul, nous voyons que le temps d'exécution moyen peut s'écrire :

$$T_m(n) = \sum_{j=1}^{j=n-1} \left(3 + 2\frac{j}{2}\right) = 3(n-1) + \frac{n(n-1)}{2} = \frac{n^2}{2} + \frac{5n}{2} - 3$$

Cette quantité est plus naturellement plus petite que $T(n)$. Néanmoins, son comportement asymptotique est le même au sens où nous l'entendons. La complexité en moyenne du tri par insertion est aussi en $O(n^2)$ (on précise généralement dans ce cas que les constantes multiplicatives sont plus petites ($\frac{1}{2} < 1$))
Nous disposons maintenant d'un moyen d'évaluer les algorithmes, et donc de les comparer. Les valeurs de complexité les plus répandues sont (elles sont ici classées de la meilleure à la pire) :
– $O(1)$ pour un algorithme en temps constant ;
– $O(\ln(n))$ pour un algorithme en temps logarithmique ;
– $O(n)$ pour un algorithme linéaire ;
– $O(n \ln(n))$;
– $O(n^2)$ pour un algorithme en temps quadratique ;
– $O(n^m)$ pour un algorithme polynomial (le temps quadratique en est un cas particulier) ;
– $O(c^n)$ pour un algorihtme exponentiel (c est constant).

3.2 Complexité d'un problème

De la même façon qu'on étudie la complexité d'un algorithme, on peut étudier la complexité d'un problème donné, et parfois indiquer que le meilleur algorithme possible pour un problème donné ne pourra pas être mieux que linéaire, ou quadratique, par exemple.
Il est par exemple évident qu'un algorithme de tri ne pourra pas être mieux que linéaire, puisqu'il devra au minimum examiner chaque donnée et qu'il y en a n.

3.3 Tri rapide

Puisque le tri par insertion est en $O(n^2)$ est que la complexité minimum du problème est linéaire. On peut se demander s'il n'existe pas de meilleurs algorithmes de tri que le tri par insertion.
La méthode de tri rapide (qui fonctionne sur le principe du «diviser pour régner» à l'instar du tri par fusion) est donnée par l'algorithme 2.7 (la procédure echanger n'est pas détaillée).

ALG. 2.7 Tri rapide

```
rapide(t[] : tableau d'entiers, p, r : entiers)
    q : entier
    si p<r alors
        q←partitionner(t,p,r)
        rapide(t,p,q)
        rapide(t,q+1,r)

partitionner(t[] : tableau d'entiers, p, r : entiers)
    cle,i,j : entiers
    cle←t[p]
    i←p-1
    j←r+1
    répéter indéfiniment
        j←j-1
        répéter tant que t[j]>cle
            j←j-1

        i←i+1
        répéter tant que t[i]<cle
            i←i+1

        si i<j alors
            echanger(t,i,j)
        sinon
            renvoyer j
```

L'algorithme de tri rapide étant récursif, les temps de calcul vont s'exprimer par récurrence. Nous allons nous intéresser uniquement au cas moyen. Dans le cas moyen, on suppose que la fonction partitionner partitionne la tranche de tablea p..r en deux tranches de taille égale. Il est facile de voir que la fonction partitionner est en $O(n)$ si n est la taille de la tranche de tableau qui lui est envoyée (en effet, l'indice i croit toujours, l'indice j décroit toujours, jusqu'à ce qu'ils se croisent).
La relation de récurrence qui donne le temps d'exécution dans le cas moyen est donc :

$$T(n) = 2T(n/2) + O(n)$$

La récurrence est assez facile à résoudre si nous supposons que $n = 2^k$ avec k entier et si nous remplaçons simplement $O(n)$ par n. Nous admettrons que le résultat se généralise. On a alors :

$$T(2^k) = 2T(2^{k-1}) + 2^k$$

c'est à dire :

$$\begin{aligned} T(2^1) &= 2T(2^0) + 2^1 \\ T(2^2) &= 2T(2^1) + 2^2 = 2^2T(2^0) + 2^2 + 2^2 \\ T(2^3) &= 2T(2^2) + 2^3 = 2^3T(2^0) + 2^3 + 2^3 + 2^3 \end{aligned}$$

Il semblerait que l'on ait :

$$T(2^k) = 2^k T(2^0) + k2^k$$

On peut aisément le montrer par récurrence.
Si nous revenons à n, et en constatant que $T(1) = O(1)$, nous obtenons :

$$T(n) = n + n\ln(n) = O(n\ln(n))$$

Le temps d'exécution en moyenne du tri rapide est donc en $O(n\ln(n))$, ce qui est beaucoup mieux que le tri par insertion.
Un calcul rapide nous permettrait de vérifier que dans le cas de pire, le tri rapide a par contre un temps d'exécution en $O(n^2)$ (la différence se fait au moment où l'on considère que partitionner ne partitionne plus en deux ensembles de longueur égale, mais en un tableau de taille n et un tableau de taille 1).

Chapitre 3

Programmation en C

Le contenu de ce chapitre est désormais accessible en ligne sur la plate-forme Updago :
`http ://updago.univ-poitiers.fr`
Le cours Updago est intitulé : Algorithmique et Programmation 2

Annexe A

Travaux dirigés

TD I : Pointeurs, listes, arbres et graphes

■ Listes chaînées d'entiers

Question 1 – Longueur de liste*

Écrivez une fonction qui renvoie le nombre d'éléments d'une liste.

Question 2 – Suppression dans une liste*

Écrivez une fonction qui supprime l'élément de tête d'une liste.

Question 3 – Égalité*

Écrivez une fonction qui indique en renvoyant 0 ou 1 si deux listes contiennent les mêmes données dans le même ordre (si c'est le cas, la fonction renverra 1)

Question 4 – Suppression d'un élément

Écrivez une fonction qui supprime un élément d'une liste (on donne sa valeur en paramètre). Donnez une version qui supprime uniquement le premier élément trouvé, puis une qui supprime tous les éléments correspondants.

Question 5 – Concaténation

Écrivez une fonction qui à partir de deux listes, crée une troisième liste, concaténation de la première avec la seconde.

Question 6 – Listes triées

Écrivez une fonction qui insère un élément dans une liste triée, en conservant l'ordre de tri (croissant) de cette liste.

■ Arbres binaires

Question 7 – Affichages infixés et suffixés*

Écrivez les algorithmes d'affichage infixé et suffixé d'un arbre binaire d'entiers.

Question 8 – Taille d'un arbre*

Écrivez une fonction qui renvoie la taille (le nombre de nœuds) d'un arbre binaire.

Question 9 – Hauteur d'un arbre*

Écrivez une fonction qui renvoie la hauteur d'un arbre, c'est à dire la longueur, en nombre de nœuds, de la plus longue branche.

Question 10 – Nombre d'occurences

Écrivez une fonction qui indique le nombre d'apparitions d'un élément dans un arbre binaire.

Question 11 – Insertion dans un arbre de recherche

Écrivez une fonction qui insère un nouvel entier dans un arbre binaire de recherche.

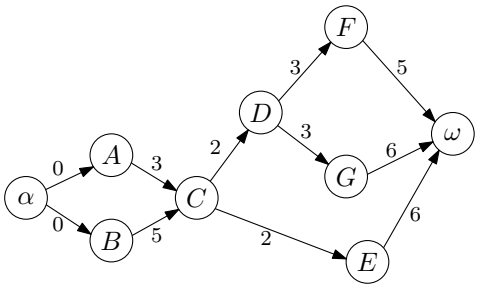


FIG. A.1 – Graphe d'ordonnancement de tâches

Question 12 – Minimum

Écrivez une fonction qui renvoie le plus petit élément d'un arbre binaire de recherche.

Question 13 – Affichage ordonné

Écrivez une procédure qui affiche dans l'ordre croissant les éléments d'un arbre binaire de recherche.

■ Graphes

Question 14 – Valeur d'arête*

Écrivez une fonction qui à partir de la représentation par listes d'adjacence indique la valeur de l'arête entre deux nœuds donnés en paramètres.

Question 15 – Nombre d'arêtes

Donnez un algorithme qui détermine le nombre d'arêtes d'un graphe orienté à partir de sa matrice d'ajacence. Et si le graphe n'est pas orienté ?

Question 16 – Parcour d'un graphes en profondeur d'abord*

Écrivez un algorithmes de parcours en profondeur d'un graphe. Un parcours en profondeur consiste à explorer d'abord un chemin le plus long possible issu d'un nœud. On utilisera une représentation des graphes par listes d'adjacence.

Question 17 – Plus long chemin

Écrivez un algorithme de recherche des plus longs chemins d'un nœud source vers tous les autres nœuds dans un graphe valué (à valeurs positives) orienté sans cycle.

Question 18 – Ordonnancement

Le graphe de la figure A.1 est un graphe d'ordonnancement de tâches.
C'est un graphe orienté, valué (à valeurs positives), et sans cycle. Chaque nœud étiqueté par une capitale romaine représente une tâche (par exemple pour la construction d'une maison). Un arc du nœud i vers le nœud j signifie que l'accomplissement de la tâche i est nécessaire au démarrage de la tâche j. Les valeurs des arêtes issues d'un sommet (toutes égales) sont le temps nécessaires à la réalisation de la tâche. Les nœuds fictifs α et ω représentent le départ et la fin du travail. Exceptés ces deux nœuds, tous ont au moins un prédécesseur et un successeur.
Donnez un algorithme indiquant, à partir d'un tel graphe, le temps minimum pour réaliser l'ensemble des tâches, ainsi que la liste des tâches critiques (les tâches pour lesquels un retard pris influera nécessairement sur le temps mis à accomplir l'ensemble des tâches).

■ Allocation dynamique

Question 19 – Structure de pile

Cette question sera l'occasion de découvrir la structure de pile et de donner des portions de code.
Une pile est un ensemble de données ordonnées. Soit elle est vide, soit elle est composée d'une cellule, puis d'une pile. C'est un cas particulier de liste, pour laquelle les opérations usuelles sont *empiler* (i.e. rajouter une cellule au sommet), *dépiler* (i.e. enlever la cellule qui est au sommet), tester si la pile est vide, ou encore récupérer la valeur de la cellule qui est au sommet. Nous travaillerons ici avec des piles d'entiers.

19-1 – Donnez la définition d'une cellule, puis d'une pile, en langage algorithmique, p uis en C.

19-2 – Écrivez une fonction qui ajoute un entier dans une pile et renvoie la pile résultante. Il faudra ici utiliser la fonction d'allocation dynamique du C.

19-3 – Écrivez une fonction qui renvoie l'entier situé au sommet de la pile.

19-4 – Écrivez une fonction qui dépile la cellule située au sommet de la pile, puis renvoie la pile résultante.

19-5 – Écrivez une fonction qui vide une pile.

19-6 – Écrivez une fonction qui teste si une pile est vide.

Bibliographie

[1] Christophe Blaess. *Programmation système en C sous Linux.* Eyrolles, 2000.

[2] Jean-Pierre Braquelaire. *Méthodologie de la programmation en C.* Dunod, 2000.

[3] Alain Cardon and Christian Charras. *Introduction à l'algorithmique et à la programmation.* Ellipses, 1996.

[4] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction à l'Algorithmique.* Dunod, 1994.

[5] Brian W. Kerninghan and Denis M. Ritchie. *Le langage C – Norme ANSI.* Dunod, 2000.

[6] Robert Sedgewick. *Algorithmes en langage C++.* Pearson Education, 1999.

[7] Robert Sedgewick. *Algorithmes en langage C.* Dunod, 2001.

[8] Raymond Séroul. *Math-Info, Informatique pour Mathématiciens.* InterEditions, 1995.

[9] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL 1.2.* Campus Press, 1999.