Algorithmes avancés

Cours Licence Info Semestre 5 – Université Bordeaux 1 – année scolaire 2004/2005

1 INTRODUCTION

Objectifs du cours :

- Comment concevoir des algorithmes efficaces ?
- Comment savoir que les algorithmes conçus sont corrects et efficaces ?
- Comprendre qu'il est peu probable qu'un algorithme efficace existe.

2 PRINCIPE D'ALGORITHMES EFFICACES

2.1 DIVISER POUR REGNER

Le principe consiste à résoudre un problème en résolvant deux ou plusieurs problèmes du même type, en simplifiant le problème premier par des sous-problèmes identiques, mais simplifiés.

Attention, ce n'est pas la même chose que la programmation structurée. Ici, c'est le programme qui divise le problème et non le programmeur.

Cette méthode donne souvent des algorithmes efficaces et élégants, mais il peut parfois y avoir des conflits.

2.2 QUELQUES EXEMPLES

2.2.1 Algorithme de Tri rapide

Dans un tableau, on choisit un élément pivot (dans la méthode de base, cela consiste à prendre la premier élément du tableau).

On divise le tableau en trois parties :

- Les éléments plus petits que le pivot,
- Les éléments égaux au pivot,
- Les éléments plus grands que le pivot.

On recommence cette méthode, récursivement pour les premières (inférieur au pivot) et troisièmes (supérieur au pivot) parties.

2.2.2 Algorithme de Tri par fusion

Méthode récursive consistant à diviser les éléments en deux parties (équilibrées), trier chaque partie, puis fusionner les parties élémentaires.

2.2.3 Algorithme Min-Max

L'algorithme Min-Max consiste à trouver le minimum et le maximum d'un tableau.

S'il n'y a que deux éléments dans le tableau, il suffit de les comparer.

Sinon, on divise le tableau en deux parties (identiques et paires… ce qui implique que le tableau est de longueur n avec n pair. Voir TD 1 pour n impair) et on recommence l'algorithme (récursif). Lorsqu'on a le maximum et le minimum de deux parties, on les compare.

Cette méthode fait approximativement 3n/2 comparaisons.

3 ALGORITHMES GLOUTONS

3.1 PRINCIPE

Les algorithmes gloutons sont ceux qui trouvent une solution parmi plusieurs possibles, comparables selon un critère global. A chaque fin d'étape de l'algorithme, on choisit l'étape suivante en fonction des informations locales. La méthode gloutonne ne trouve pas forcément la meilleure solution.

3.2 EXEMPLES

3.2.1 Arbre recouvrant de poids minimum

Cet algorithme permet, dans un graphe ou les arêtes ont un poids, de trouver l'arbre qui relie tous les sommets avec une somme des poids minimale.

On peut considérer deux méthodes :

Algorithme très glouton non optimal : on commence avec un arbre d'un seul sommet et on choisit toujours l'arête de poids minimum pour rejoindre le sommet suivant.

Algorithme glouton et optimal : on commence avec n arbres (n correspondant au nombre de sommet du graphe) d'un seul sommet et on choisit toujours l'arête de poids minimum qui relie deux arbres.

3.2.2 Sac à dos continu

Le problème consiste à remplir un containeur avec une charge la plus grande possible étant donné plusieurs matériaux, chacun avec une valeur/kilo et une quantité disponible.

Il s'agit d'un algorithme glouton : il faut toujours chercher et choisir le matériau avec le meilleur rapport valeur/poids.

3.3 CAS OU LA METHODE GLOUTONNE N'EST PAS OPTIMALE

3.3.1 Exemples

On peut retenir trois exemples :

- Sac à dos discret : avec des objets que l'on ne peut découper.
- Coloration de graphe : on veut colorier les sommets d'un graphe avec le plus petit nombre couleurs possible sachant que deux sommets voisins (reliés par une arrête) doivent avoir des couleurs différentes.
- Commis voyageur : étant donnée une carte des distances entre des paires de villes, trouver le chemin le moins coûteux qui passe par chaque ville une seule fois (et retourne au point de départ).

3.3.2 Commentaires sur la non existence d'algorithmes efficaces (connus) pour ces trois exemples

Ces trois problèmes sont des variantes des problèmes NP-complets. Personne ne connaît des algorithmes garantis de trouver une solution optimale en temps raisonnable (borné par un polynôme), mais il n'est pas dit que ce soit impossible.

Donc, il est utile de considérer des algorithmes qui ont une garantie de trouver une solution proche de l'optimale. Un premier essai à un tel algorithme pourrait être un algorithme glouton simple. Un deuxième essai serait d'améliorer le résultat du premier par des modifications locales gloutonnes.

4 ARBRES

4.1 ARBRES BINAIRES DE RECHERCHE

Les arbres binaires sont des structures utiles. Exemples :

- La structure représente la structure logique des données (arbre syntaxique, arbre généalogique)
- La structure est utilisée pour permettre les opérations nécessaires sur une structure abstraite (arbre binaire de recherche, additions, suppressions d'éléments plus efficacement qu'avec un tableau ou liste avec pointeurs).

4.1.1 Insertion, recherche, annulation dans un arbre binaire de recherche

4.1.1.1 La recherche dans un arbre binaire

On commence la recherche à la racine, on parcours à gauche ou à droite selon la comparaison avec la valeur à chaque nœud (échec si sous arbre vide).

4.1.1.2 Insertion d'une valeur dans un arbre binaire

Idem que pour la recherche, en remplaçant le sous-arbre vide par un nouveau sous arbre contenant la valeur à insérer.

4.1.1.3 Suppression dans un arbre binaire

La suppression est facile au niveau d'une feuille ou si le nœud n'a qu'un seul fils. S'il y a deux fils, on remplace par exemple le nœud supprimé par le sous arbre.

4.1.1.4 Remarque

Les traitements dans la hauteur de l'arbre ont toujours un nombre d'opération linéaire. Il s'agit de quelques manipulations de pointeurs

4.1.2 Les ordres de parcours

Il existe trois ordres de parcours (souvent utiles) permettant de visiter tous les nœuds d'un arbre (binaire) :

- Ordre infixe : (par exemple, afficher les éléments d'un arbre binaire de recherche en ordre). On parcourt tous le sous arbre gauche (en parcourant les sous arbre du sous arbre dans le même ordre… récursivement), ensuite la racine, puis tout le sous arbre droit.
- Ordre préfixe : (utile pour calculer la profondeur de chaque nœud) on traite d'abord la racine, puis le sous arbre gauche et enfin le sous arbre droit.
- Ordre post-fixe : (pour calculer par exemple la valeur d'une expression à partir de son arbre syntaxique, ou pour calculer une hauteur) d'abord les deux sous-arbre (gauche et droit), et la racine pour finir.

4.2 ARBRES EQUILIBRES

Un arbre binaire de recherche peut avoir une hauteur près de son nombre de nœud (n), en revanche, la moyenne est O(log n). C'est cette hauteur qui détermine le temps des opérations de base.

Il existe un grand nombre possible d'arbres avec n nœuds, on en obtient un selon l'ordre des insertions.

L'idée, pour obtenir un arbre équilibré, consiste à faire un peu plus de travail (à chaque insertion) afin de garder une hauteur plus raisonnable (de préférence O(log n)).

4.2.1 Arbre AVL

Un arbre AVL est un arbre dont la différence entre les hauteurs des deux sous arbres est au maximum 1. Ce type d'arbre nécessite le stockage d'une petite information cachée à chaque nœud : la différence entre les deux hauteurs des sous-arbres (-1, 0, 1).

Dans ce type d'arbre, la suppression d'un élément devient donc un peu plus compliquée.

4.2.2 Arbres 2-3

Un arbre 2-3 est un arbre pour lequel chaque nœud a soit une valeur et un fils, soit deux valeurs et trois fils. L'ordre est : sous-arbre, valeur, sous-arbre [, valeur, sous-arbre].

Dans un tel arbre, toutes les feuilles sont à la même profondeur.

- Pour ajouter une valeur dans un nœud qui n'en contient qu'un : trivial.
- S'il y en a déjà deux : insérer le deuxième dans le nœud père et diviser ce nœud en deux (avec peut-être une réaction en chaîne vers la racine).

5 RECURSIVITE

5.1 IDEE

une fonction peut s'appeler elle-même exactement comme un appel d'une autre fonction. Dès l'appel récursif achevé l'exécution du programme continue (et sans effet sur les variables et paramètres de l'appel initial).

Cette méthode est valable pour les fonctions et procédures (fonctions void).

On parle aussi de récursivité mutuelle lorsque deux ou plusieurs fonctions s'appellent entre elles (donc appel avant déclaration pour quelques unes).

5.2 EXEMPLES FAVORIS

5.2.1 Calcul de factorielle

```
Fonction fact(n) : entier
        Début
                Si (n=0) retourner (1)
                Sinon retourner (n*fact(n-1))
        Fin
```

5.2.2 Fibonacci

```
Fonction Fb(n) : entier
        Début
                Si (n≤2) retourner 1
                Sinon retourner (Fb(n-1)+Fb(n-2))
        Fin
```

5.2.3 Remarques

La fonction récursive est une méthode bizarre de calculer une factorielle. Une boucle simple suffit. La fonction récursive est une très mauvaise méthode de calculer une suite de Fibonacci car le temps de calcul est exponentiel!

Donc, ne pas oublier qu'une méthode récursive est plus élégante et claire… mais il faut aussi considérer le temps de calcul, l'efficacité.

5.3 MEILLEURS EXEMPLES

5.3.1 Calcul de x^n

```
Fonction Puissance (x, n) : réel
        Début
        Selon que :
                N=1 : retourner x
                N%2=0 : retourner carrée(puissance(x,n/2))
                Défaut : retourner x*carrée(puissance(x,n/2))
        Fin
```

Remarque : on saute de x^i à x^{2i} dans une seule multiplication, ce qui est plus difficile à gérer avec une boucle. Il s'agit donc d'un bon exemple d'utilisation de fonction récursive.

5.3.2 Tour de Hanoi

5.3.2.1 Rappel du principe

Déplacer une tour de n disques de taille différente d'une colonne à une autre en utilisant une seule colonne auxiliaire, selon la règle qu'on ne peut déplacer qu'un disque à la fois et chaque colonne a toujours ses disques en ordre décroissante de taille.

5.3.2.2 Algorithme

```
Procédure Hanoi(n {nombre de disques}, a, b, c {tours})
        Début
                Si n=0 alors exit
                Hanoi(n-1,a, c, b)
                Afficher(a vers b)
                Hanoi (n-1, c, b, a)
        Fin
```

L'appel initial peut être : Hanoi(64, 1, 2, 3)

5.3.3 Autres exemples

- Affichage en (disons) binaire : une boucle simple affiche les chiffres dans l'ordre inversé
- Boucles imbriquées de profondeur inconnue
- Algorithmes « diviser pour régner »

5.4 PARAMETRES ET VARIABLES LOCALES

Les paramètres et les variables déclarées localement dans une fonction (si elle est récursive ou non) sont créés au moment de l'appel/activation de la fonction et continuent à exister jusqu'à la sortie finale de l'appel.

Ils sont inaccessibles et immuables pendant d'autres appels imbriqués (sauf utilisation de pointeurs...)

Donc, quand une fonction s'appelle récursivement, à un moment de l'exécution du programme, il existe un nombre indéfini d'occurrences de ses paramètres et variables, mais, en général une seule occurrence accessible. Il y a possibilité d'échec parce que l'espace mémoire ne suffit pas pour toutes ces variables (segmentation fault en C).

5.5 EXEMPLES UTILISANT DES TABLEAUX

5.5.1 Listes multiplicatives

```
Procédure Suites(n, m, T) :
        Début
                Afficher (n, T[n], T[n-1], …,T[1])
                T[m+1]<- n
                Pour i de 1 à n/2 faire :
                        Si n%i=0 Suites(i, m+1, T)
        Fin
```

5.5.2 Recherche d'un parcours de l'échiquier par un cavalier (et revenir au début)

```
Procédure Cavalier (i, j, n, T) {i, j : position – T tableau à double entrée – n : nombre de coups/cases parcourues}
        Début
                Selon que :
                i≤0 ou j ≤0 ou i>8 ou j>8 : exit
                i=1 et j=1 et n=65 afficher FIN
                cas T[i,j]>0 exit {Si =0 : alors la case n'a pas encore été parcourue}
                Défaut : T[i, j]<-n
                        Cavalier (i-2, j-1, n+1, T)
                        …
        Fin
```

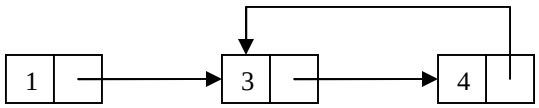
6 STRUCTURES RECURSIVES

6.1 DEFINITION

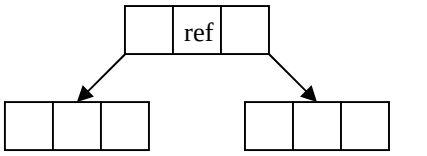
On appelle structure récursive une structure qui contient un (des) pointeur(s) cers une structure de même type.

6.1.1 Exemple 1 : Liste

Type Liste
enreg
    Val : entier
    Queue : ^Liste
Fin enreg



6.1.2 Exemple 2 : Arbre



(sous) Arbre gauche (sous) Arbre droit

6.1.3 Remarque

Pour traiter des structures récursives on utilise principalement des fonctions récursives

6.2 EXEMPLES D'ALGORITHMES AVEC LES LISTES

6.2.1 Longueur

Fonction permettant de calculer la longueur (parcours en longueur) d'une liste.

```
Fonction longueur (val P : ^Liste) : entier
        Si P=NULL alors retourner 0
        Sinon retourner (1+Longueur(P.^queue))
```

6.2.2 Somme

Fonction permettant de retourner la somme des valeurs contenues dans la liste. Le principe est le même que pour la fonction Longueur avec : « retourner (P.^val+somme(P.^queue)) ».

6.2.3 Concaténer

Cette fonction permet de concaténer une liste Q à une liste P (on va chercher la fin de P pour y ajouter Q).

```
Procedure concaténer (ref P, Q : ^Liste)
        Si P=NULL alors P :=Q
        Sinon concaténer (P^queue, Q)
```

6.2.4 Fusionner deux liste triées

```
Fonction fusionner (val : P, Q : ^Liste) : ^Liste
        Var temp : ^Liste
        Selon que :
                Cas P=NULL retourner Q
                Cas Q=NULL retourner P
                Cas P.^val < Q.^val
                        Allouer (Temp)
```

```
            Temp^.val=P^.val
            Temp^.queue:=P^.queue
            Temp^.queue :=Fusionner(P^.queue, Q)
            Retourner Temp
Sinon
            … (idem dans l'autre sens)
```

7 PROGRAMMATION DYNAMIQUE

La programmation dynamique est une méthode générale de calcul de plusieurs valeurs dont la plupart dépendent d'autres. Elle est utilisée dans les cas où un algorithme simple naïf serait trop coûteux. Elle consiste à appliquer le principe évident qu'il est inutile de calculer la même chose deux (ou plusieurs) fois. *Pour cela, on va stocker les résultats déjà calculés dans un tableau.*

7.1 EXEMPLES

7.1.1 Un TRES petit exemple: la suite de Fibonacci

Nous avons vu que la méthode simple récursive n'est pas bonne. Une meilleure méthode consiste à stocker les résultats déjà calculés dans un tableau et ce, jusqu'à la valeur souhaitée :

1	1	2	3	5	…

7.1.2 Les coefficients binomiaux

Le nombre de façons de choisir m objets parmi n. On connaît tous la formule $\frac{n!}{m!(n-m)!}$. Une autre méthode consistant à voir si on laisse le premier objet , ou on le prend donne la relation suivante qui pourra être facilement implémentée à l'aide d'un tableau : $Bin(n,m)=Bin(n-1,m)+Bin(n-1,m-1)$ (sauf dans les cas n=m ou n=0).

7.1.3 Le nombre de partitions d'un entier

Supposons que l'on cherche le nombre de façons de diviser n objets identiques en (1 ou) plusieurs parties. Ce calcul peut paraître peu évident car il n'y a pas de récurrence directe. Mais si on ajoute un deuxième paramètre (la taille de la plus grande partie) on trouve une récurrence dont les résultats pourront être stockés : $Part(n,max)=\sum_{i=1}^{max} Part(n-max,i)$ sauf dans les cas $n \leq max$.

7.1.4 Remarque intermédiaire

Une méthode naïve aurait été de calculer directement la récurrence (peut-être par une fonction récursive) *sans utiliser de tableau*. Le temps de calcul aurait été déterminé par le nombre de feuilles dans l'arbre de tous les calculs possibles. On peut arriver à des algorithmes qui mettent des siècles à calculer. La programmation dynamique peut réduire la durée de ces calculs à une fraction de seconde.

7.1.5 Un exemple (un peu) moins mathématique faire l'appoint

Je veux calculer le nombre de façons de faire un total de N centimes étant donné le nombre de types de pièces qui existent (n), le nombre que je possède et la valeur en centimes de chaque type (tableaux Nombre et Valeur indicés de 1 à n) .

Calcul de Nombre(Total,i) : le nombre de façons de faire un total de Total centimes en n'utilisant que les pièces des i premiers types :

```
t:=0;  // sera le nombre souhaite
u:=0;  // le nombre de pieces de type i utilisees
repeter
```

t:=t+Nom(Tot-u*Valeur[i],i-1);
u:=u+1;
jusqu'a u>Nombre[i] ou u*Valeur[i]>Tot
finrepeter;
(sauf les cas i=0 (Nombre(Total,0)=1 ou 0 selon si Total=0)

7.1.6 Un exemple avec temps, probabilité et stratégie: lancer des dés pour finir avec 100 points.

On lance des dés 20 fois. A chaque coup, on gagne le nombre de points sur la face visible du dé. On veut finir avec exactement 100 points. Le calcul consiste à déterminer la probabilité de réussir.

Pour ne pas faciliter le problème, on peut, à chaque coup, choisir de lancer un ou deux dés (de façon à maximiser la probabilité de réussite).

On calcule pour chaque p (nombre de points entre 0 et 100) et chaque c (nombre de coups entre 0 et 20) la probabilité de gagner si on veut encore p points après le c-ème coup, disons P(p,c) (sauf si c=20) :

- Si on lance un dé : probabilité $=\sum_{i=1}^{6} P(p-i,c+1)/6$
- Si on lance deux dés: probabilité $=\sum_{i=2}^{12} P(p-i,c+1)\times Prob[i \text{ points des 2 dés}]$
- Et on prend le maximum des deux comme valeur de P(p,c).
- Avec correction pour les cas où on essaie de calculer une valeur avec $p < 0$!

7.1.7 Le problème du sac à dos discret

Etant donné un sac à dos et n objets de poids et valeur différents, quel est la valeur maximale d'un ensemble d'objets qui ne dépasse pas la capacité (en poids) du sac?

Supposons que l'on considère les objets en ordre et décide, pour chacun, si on le prend ou non. Quand on considère l'objet numéro i, la valeur de ce que l'on pourrai choisir parmi (i+1, n) dépend de la partie non utilisée de la capacité. On calcule donc pour chaque i et chaque capacité (entre 0 et la capacité initiale) une valeur maximale.

7.2 REMARQUE 1 : ECONOMISER L'ESPACE

Quand la récurrence donne la valeur de la colonne i du tableau comme fonction de la colonne i+1 (ou i–1) on n'est pas obligé de garder tout le tableau. Il suffit de garder deux colonnes à chaque moment (vrai pour les algorithmes de probabilités dans le graphe et le sac-à-dos par exemple)

Si, en outre, chaque élément d'une colonne ne dépend que des éléments de l'autre colonne qui sont en dessus (ou en dessous), il suffit de garder une colonne qui, à chaque moment, peut contenir une partie de l'ancienne colonne et une partie de la nouvelle. (vrai pour le sac-à-dos).

7.3 REMARQUE 2 : EVITER LES CALCULS INUTILES

Il se peut que quelques éléments du tableau ne soient pas utilisés dans le calcul final qui nous intéresse (c'est-à-dire ni directement ni indirectement) mais que la structure du problème soit trop irrégulière pour savoir à l'avance lesquels (par exemple dans le cas de la chaîne de Markov, il est inutile de calculer la probabilité de réussir en 90 transitions à partir de l'état j s'il n'est pas possible d'arriver à cet état en 10 transitions de l'état initial).

Dans ce cas, la programmation dynamique comme décrite fait des calculs inutiles qui n'auraient pas été faits par la méthode récursive citée avant. Mais il est toujours vrai que cette méthode récursive fait BEAUCOUP plus de calculs parce qu'elle fait très souvent le même calcul plusieurs fois.

Il existe une méthode mixte qui ne fait que les calculs utiles et ne les fait qu'une fois: elle consiste à utiliser la structure récursive et un tableau; la première fois qu'un résultat est calculé il est inséré dans le tableau; les fois suivants, cette valeur est récupérée du tableau.

8 STRUCTURES COMPLEXES

8.1 EXEMPLES

- Sous-ensembles d'un ensemble
- Arbres (binaires ou autres...)
- Graphes
- Permutations

Si un programme construit et utilise une seule structure, des méthodes ad hoc suffisent. Si le programme en utilise beaucoup, il faut penser à l'efficacité des opérations nécessaires.

8.2 OPERATIONS UTILES

- Boucler sur toutes les structures
- Stocker une structure dans l'espace minimum possible (code)
- Récupérer une structure stockée (décode)
- compter les structures

Remarque : Il faut souvent choisir une taille de structure (disons n), sinon on ne peut utiliser de fonction de boucle.

8.3 SOUS-ENSEMBLES ET PERMUTATIONS

Pas de gros problème :

- Sous-ensemble de n objets entier en 0...2n-1
- Permutation? considérer une permutation comme n entiers en 0...n-1 donne une représentation par un entier en $0...n^n-1$
- Mais tous les entiers ne correspondent pas à une permutation (n^n entiers mais n! permutations)
- Interpréter suite a_1, a_2, ..., a_n avec $1 <= a_i <= i$: échanger les éléments n et a_n, ensuite n-1 et a_n-1 et ainsi de suite; chaque permutation est générée une seule fois.
- Facile à générer la suite des a à partir d'un entier en 0...n!-1
- (Remarquer existence d'algorithmes plus astucieux qui génèrent toutes les permutations avec une seule transposition à chaque itération)

8.4 EXEMPLE SUR LES ARBRES

8.4.1 Générer tous les arbres binaires (sans valeurs aux sommets) (de taille n)

Pour toute taille i possible du sous-arbre gauche (0...n-1)
    générer tous les sous-arbres gauches (taille i)i
    générer tous les sous-arbres droits (taille n-1-i)

- mais complication de programmation; il faut chaque combinaison possible d'un sous-arbre gauche et droit
- une solution: procédures init et prochain

8.4.2 Compter les arbres (binaires)

En ce cas il existe une formule simple mais...

- Soit T(n) le nombre d'arbres de taille n T(0)=1
- Considérer le nombre avec sous-arbre gauche de taille i : $T(n)=\sum_{i=0}^{n-1} T(i)*T(n-1-i)$
- Programmation dynamique!

8.4.3 Coder un arbre (de taille n, entier en 0...T(n)-1)

- Ordonner les arbres selon la taille de leur sous-arbre gauche, et ensuite selon son code

- code(A)=sommei=0taille(A.gauche)-1T(i)*T(n-1-i)+code(A.gauche)*T(A.droite)+code(A.droite)
- code(arbre.vide)=0

8.4.4 Décoder un arbre

Etant donné le code c d'un arbre de taille n, trouver :

- La taille du sous-arbre gauche
- Le code du sous-arbre gauche
- Le code du sous-arbre droit
- Et continuer récursivement

8.4.5 En général

Un algorithme de génération donne des algorithmes de codage décodage :

- Ordonner les objets selon les choix faits par l'algorithme
- considérer le nombre d'objets susceptibles d'être générés par chaque choix
- ajouter pour coder
- soustraire pour décoder

8.4.6 Des cas plus compliqués

Par exemple des arbres (non binaires), par exemple : Arbre = vide ou racine + ensemble de sous-arbres. *Il s'agit d'ensemble, donc il n'y a pas d'ordre.*

Pour simplifier, on considère les cas de degré max 2. Un arbre peut être alors représenté par un arbre binaire mais, de plusieurs façons!

8.5 REPRESENTATIONS CANONIQUES

Quand une structure a plusieurs représentations, on en choisit arbitrairement une comme la ``vraie'' ou canonique. Par exemple ordonner les arbres (binaires) par taille, sous-arbre gauche, sous-arbre droit et dire que la représentation canonique d'un arbre est le premier arbre binaire qui en est une représentation.

Pour cela, on utilise une fonction qui permet de décider si une représentation est canonique :

Récurrence pour le nombre d'arbres de taille n (n > 1):

$$G(n) = G(n-1) + \sum_{i=1}^{\frac{n}{2}-1} G(i) * G(n-1-i) + (\text{si n impair})G((n-1)/2) + (G((n-1)/2)/2) .$$

Remarques :

- arrondir vers le bas dans la division n/2
- les quatre parties sont: un seul sous-arbre (forcément de taille n-1, deux sous-arbres de taille différente, le plus petit de taille i, deux sous-arbres identiques, deux sous-arbres différents de la même taille).
- Mêmes principes pour codage et décodage

8.6 DES PROBLEMES DIFFICILES

Les problèmes de codage et décodage sont parfois très compliqués. On peut citer comme exemple les graphes sur n sommets de degré maximum (par exemple) 3. On peut les générer en considérant chaque sommet à son tour et ajoutant un nombre d'arêtes jusqu'à 3 - degré actuel. Mais contraintes : pas d'arête à un sommet qui a déjà comme degré 3. Donc, on se retrouve avec une multitude de possibilités de compléter?

8.7 GRAPHES ET ISOMORPHISME!

Souvent, on considère deux structures comme identiques si on peut transformer l'une dans l'autre par une permutation des composants. On veut générer (ou compter) un représentant canonique de chaque classe d'équivalence. Mais les classes ne sont pas toutes de la même taille! Il est donc difficile de savoir si une structure est canonique ou de décider si deux structures sont isomorphes (Pb NP-complet).

9 FONCTIONS SUR LES NOMBRES

9.1 LES NOMBRES ALEATOIRES

Il est souvent utile d'avoir une fonction qui produit des nombres apparemment aléatoires. La fonction rand() prédéfini peut faire se genre d'opération. A chaque fois qu'elle est appelée, elle donne un nombre différent (normalement en [0..N]). En réalité, elle n'est pas vraiment aléatoire mais on considère statistiquement que le résultat est convenable.

9.2 GENERATION DE NOMBRES

L'algorithme suivant permet de générer des entiers de manière pseudo-aléatoires. Soient x1,...,xi,... :

- On effectue un calcul simple comme xi+1 = xi*m + c (mod N) pour m bien choisi et N une puissance de 2.
- On divise par N pour un réel aléatoire en [0..1]
- On multiplie ensuite par R pour un réel en [0..R]
- Et on arrondit pour un entier en [0..R?1]

Attention toutefois : prendre xi mod. R est très mauvais si R est une puissance de 2!!

9.3 GENERATION DE STRUCTURES

La génération de structure consiste :

- Soit à générer un entier dans un bon intervalle et le décoder
- Soit à générer la structure itérativement avec un entier généré chaque fois que l'algorithme doit prendre une décision .

La deuxième méthode plus facile à programmer et évite problèmes d'entiers trop grands.

9.3.1 Utilisation 0: interface avec l'extérieur

- programme de jeu: pour ne pas toujours jouer pareil
- joli écran
- réseau: attendre un temps aléatoire avant de réessayer un transfert après une collision.

9.3.2 Utilisation 1: tester un algorithme

- Générer un jeu de tests sur les cas aléatoires (Ne remplace pas les tests systématiques qui utilisent chaque branche du programme)

Attention, se rappeler que :

- « Tester ne prouve jamais qu'un programme est sans fautes »
- Sauf pour ceux avec un nombre fini (et petit) de données possibles

9.3.3 Utilisation 2: calcul numérique de statistiques

On peut donner plusieurs exemples d'applications dans ce domaine :

- Calculer la superficie moyenne du polygone défini (enveloppe convexe) par 10 points choisis aléatoirement dans le carré [0,1][0,1].
- Générer, disons 1000, cas aléatoires et calculer la superficie de chacun

Attention, cette utilisation peut être :

- dangereuse : car très faible probabilité qu'on soit loin du vrai moyen.
- très dangereuse : si la distribution est très bizarre (écart type très grand), presque certain d'être loin du vrai moyen.

9.3.4 Utilisation 3: Simulation de processus (naturels ...)

Exemples:

- Propagation d'une maladie
- Effet sur la circulation si l'on modifie une route

- Résilience d'un réseau à des pannes
- Comportement d'une bourse où des commandes de ventes ou d'achats sont générées par des logiciels

En fait, cette technique peut être utilisé partout où la complexité des interactions entre les composants d'un système exclut tout calcul direct.

9.3.5 Utilisation 4: algorithmes probabilistes

Quelques exemples :

- Algorithmes de calcul exact qui utilisent des choix aléatoires
- Algorithmes calculant une probabilité d'échec (mais, souvent, ils reconnaissent qu'un échec est arrivé). Donc, répéter plusieurs fois un évènement (avec choix différents) réduit exponentiellement la probabilité d'échec

10 COMPLEXITE DES ALGORITHMES

10.1 COMPLEXITE DES ALGORITHMES : TEMPS ET ESPACE

Le plus souvent le temps de calcul est le plus important. Il faut considérer le temps et l'espace dans le pire des cas, c'est-à-dire définir un majorant de la complexité en temps et en espace. Pour cela, on effectue une analyse de l'algorithme (analyse souvent facile, souvent difficile et quelquefois impossible).

10.2 COMME FONCTION DE LA TAILLE DES ENTREES NOTATION O( )

Le temps (ou espace) dépend des entrées. On n'a donc pas de majorant universel. Il faut considérer le temps comme fonction de la taille n des entrées, en nombres, bits, caractères et chercher à majorer cette fonction. On obtient une complexité souvent exprimée t(n)=O(f(n) c'est-à-dire il existe N et c tels que si n > =N, alors t(n) < =cf(n).

On dit aussi omicron, omega, OMEGA, THETA.

10.3 COMPLEXITE D'UN ALGORITHME, PAS D'UN PROBLEME

Ici, nous considérons l'analyse du temps d'un algorithme donné. La complexité d'un problème peut être défini (approximativement) comme la complexité du meilleur algorithme possible qui le résout. L'analyse d'un algorithme fournit un majorant de la complexité du problème.

Trouver un minorant est, le plus souvent, beaucoup plus difficile ...

10.4 ANALYSES

10.4.1 Analyses simples du texte du programme

- Une boucle 1 à n prend le temps O(n)
- Deux boucles imbriquées 1 à n prennent temps $O(n^2)$, et ensuite de suite.
- Une recherche dichotomique sur n possibilités prend le temps O(logn)
- Algorithme d'Euclide pour pgcd(m,n) (supposons m > =n); m remplacé en 2 itérations par m mod n prend dans le pire des cas $[(m-1)/2]$ c'est-à-dire O(logn). Mais est-ce vraiment possible ?

10.4.2 Analyses plus complexes du comportement du programme

On utilise ce type d'analyse dans le cas des boucles où le nombre d'itérations n'est pas évident. On peut notamment citer les exemples suivants :

- chercher quelque chose: s'il y a une borne évidente sur le nombre de cas à considérer, ça donne un majorant mais peut-être très grossier
- processus itératif qui converge, peut-être, très lentement
- boucles très simples mais où le calcul du nombre d'itérations semble très difficile

Par exemple l'algorithme suivant :

Tant que (n>1)
si (n~pair) alors n:=n/2;
sinon n:=3n+1
fin si
fin tant que
Personne ne peut garantir que ce programme s'arrête pour tout n positif!

10.4.3 Analyses par récurrence

Prenons par exemple le problème 3SAT : expression booléenne qui est la conjonction de *clauses* de la forme (u ou v ou w) où u v et w sont des variables ou négations de variables. *L'algorithme brut consiste à considérer les 2^n affectations possibles des n variables. Dans le cas ou n=3, la manière la plus efficace consiste à diviser l'algorithme en trois sous problèmes* :

- u vrai,
- u faux mais v vrai,
- u et v faux mais w vrai

La taille des sous-problèmes est donc respectivement $n-1, n-2, n-3$. Le majorant est alors donné par $t(n)=t(n-1+t(n-2)+t(n-3)$ c'est-à-dire $t(n)=O(\alpha^n)$ où α est la (plus grande des) racine(s) de $x^n=x^{n-1}+x^{n-2}+x^{n-3}$, ou $1=x^{-1}+x^{-2}+x^{-3}$ (< 2).

Mais ce n'est pas optimal.

10.4.4 Les problèmes NP-complets

- Le meilleur algorithme connu est THETA(2^n) (SAT)
- Rarement n! (isomorphisme de (sous-)graphes)
- Ou α^n avec $\alpha < 2$
- Ou $2^{racine(n)}$

10.4.5 Analyse pire cas, en moyenne, ou amortie

- Pour un algorithme seul, l'analyse dans le pire des cas est (probablement) la plus importante
- pour une fonction qui va être appelée plusieurs fois par un algorithme plus grand, c'est peut-être la moyenne du nombre d'appel
- Mais ceci ne peut donner qu'un majorant probabiliste pour l'algorithme
- Si on peut démontrer que le pire comportement de la fonction ne se produit que rarement dans le fonctionnement du programme, cela donne un majorant déterministe *amorti*
- par exemple, une fonction de mise à jour d'une structure de données

11 ETUDE DU FONCTIONNEMENT D'UN PROGRAMME

11.1 "DEMONTRER" QU'UN PROGRAMME A L'EFFET SOUHAITE

Pour démontrer qu'un programme à l'effet souhaité, on peut :

- *Ajouter des " assertions" valables à différents points du programme,* certaines évidentes et d'autres des conséquences de celles déjà connues, etc.
- Vérifier que les conditions désirées sur le résultat du programme sont aussi des conséquences de certaines assertions .

Pour une suite d'instructions sans boucles c'est simple, même avec instructions conditionnelles, ...

11.1.1 Exemple: échanger deux variables

Soit le code t:=x; x:=y; y:=t;

On peut fractionner le fonctionnement du programme de la manière suivante :

- Au début $x=x_0$ et $y=y_0$

- après la première affectation: $t=x_0$ et $y=y_0$
- et la deuxième: $t=x_0$ et $x=y_0$
- et à la fin: $y=x_0$ et $x=y_0$

Règle de déduction : après l'affectation x:=E, assertion A(x) est vraie si A(E) était vraie avant

11.1.2 Exemple: échange conditionnelle

Soit le code si x>y alors début t:=x; x:=y; y:=t;fin

On peut en déduire le déroulement suivant :

- après le début on a une nouvelle assertion garantie vraie: x > y
- d'où on peut déduire que avant le fin on a y > x
- implicitement sinon ; ne rien faire
- après le sinon on a x < =y

Donc x < =y est vrai à la fin de chaque branche et x < =y est vrai à la fin de l'instruction conditionnelle.

11.1.3 Les boucles

Pour vérifier le bon fonctionnement des boucles, il faut prévoir et vérifier :

- Un ensemble d'assertions connues avant la boucle (la précondition)
- Un ensemble d'assertions souhaitées après la boucle (la postcondition)
- Ajouter d'autres assertions vraies à chaque itération

L'astuce est de trouver les bonnes nouvelles assertions

11.2 DEMONTRER QU'UNE CONDITION EST VRAIE APRES UNE BOUCLE

Prendre l'exemple d'une boucle :

```
tant que ExpressionBooleenne faire
        BlocInstruction
fintantque;
```

Trouver une assertion ``invariante'' telle que

- La pré-condition doit entraîner l'invariant au début de la première itération ;
- Et l'invariant Plus la ExpressionBooleenne entraînent que l'invariant est vrai à la fin de l'itération ;
- Et l'invariant Plus la négation de ExpressionBooleenne entraînent la post-condition .

11.2.1 Un exemple très simple

la somme des éléments d'un tableau T[1..n]:

```
i:=1;
s:=0;
tant que i<=n faire
    s:=s+T[i];
    i:=i+1;
fintantque;
```

L'étude de cette boucle se fait ainsi :

- L'invariant? s= somme des éléments de T de T[1] jusqu'à T[i-1]
- vraie chaque fois qu'on calcule l'expression booléenne (mais pas juste après la première affectation!)
- pré-condition i=1 et s=0
- pour déduire la post-condition souhaitée, ajouter à l'invariant $i \leq n+1$

11.2.2 Terminaison des boucles

- La méthode donnée peut démontrer que la post-condition est vraie quand et si l'exécution d'une boucle termine.

- Il faut aussi démontrer la terminaison.
- définition d'une mesure du calcul qui reste à faire (p. ex. n+1-i)
- Qui diminue à chaque itération ?
- La boucle termine dès qu'elle est nulle (ou négative)
- une valeur entière !

11.2.3 Un (mauvais) algorithme de tri

Le code suivant est un mauvais algorithme de tri :

```
i:=1;
tant que i<n faire
        si T[i]>T[i+1]
                alors debut ???;i:=1; fin
        sinon i:=i+1;
        finsi
fintantque
```

On peut démontrer qu'il s'agit d'un mauvais algorithme de tri :

- si on démontre que le ??? ne change ni l'ensemble des valeurs de T (mettons val(T)), ni i, on prend pour invariant val(T)=val(T_0) et T[1..i] trié; et démontre que la boucle trie T (si elle termine)
- et si ??? échange T[i] et T[i+1], on peut démontrer aussi que la boucle termine
- mesure: n×nombre d'inversions+n–i

11.2.4 Autres exemples

- recherche linéaire
- recherche dichotomique dans un tableau trié
- calcul des puissances
- pgcd (généralisé)

11.3 DES CAS OU LE BON INVARIANT AIDE A CONCEVOIR L'ALGORITHME?

- recherche d'une valeur dans un tableau de deux dimensions où chaque ligne et chaque colonne est triée
- recherche dichotomique dans chaque ligne prendrait temps n log(n);
- comparer la valeur cherchée avec un élément au milieu du tableau n'aide pas
- comparer avec un élément dans un coin permet de supprimer une ligne ou une colonne

Un invariant peut servir à concevoir, expliquer, documenter, comprendre, tester, un algorithme

11.3.1 Exemple : Trouver le vainqueur d'une élection

- Un tableau V (unidimensionnel) contient des votes; étant donné que quelqu'un a une majorité absolue, trouver qui dans une lecture du tableau!
- Invariant: Dans la section V[1..x], personne n'a pas de majorité absolue; dans la section V[x+1..i], candidat c a m votes de plus que tous les autres candidats (m > 0);
- facile à trouver le code qui maintient l'invariant
- ajouter une dernière phase de vérification si besoin

11.4 LES PROCEDURES ET FONCTIONS

Démontrer le bon fonctionnement d'une procédure ou d'une fonction consiste à :

- *Démontrer pour chaque fonction () qu'elle produit le résultat souhaitée* ;
- *En utilisant ce même fait pour les autres fonctions déjà prouvées* ;
- *Pour les fonctions récursives, un raisonnement par récurrence* (qui nécessite souvent une description minutieuse du résultat dans tous les cas).

12 LE PROBLEME DE FUSION-RECHERCHE

Soit un nombre n d'objets (disons les entiers 1...n). Au départ chaque objet constitue un ensemble.

Considérons deux opérations: FUSIONner deux ensembles et CHERCHer dans quel ensemble se trouve un objet donné (on peut supposer que l'on est libre de choisir comme nom d'un ensemble produit le nom qui convient parmi les deux noms des ensembles fusionnés).

Quel est le temps de faire une suite d'opérations?

12.1 QUELQUES METHODES

12.1.1 Une Méthode naïve: temps linéaire

Quand on fait la fusion de A et B, choisir comme nom celui de A et parcourir B en changeant le nom associé à chaque élément de B. Cette méthode nécessite que les éléments de chaque ensemble soient reliés (par exemple dans une liste avec pointeurs). Dans le pire cas on fusionne toujours un A d'un seul élément avec un grand B.

On obtient donc un temps n−1 pour la dernière fusion. Et la moyenne dépend de l'application.

12.1.2 Une astuce simple: temps (amorti) O(logn)

IL faut toujours choisir comme nom principal celui de l'ensemble le plus grand (ce qui nécessite le stockage de la taille de chaque ensemble. Cette taille est facile à mettre à jour après une fusion).

Remarque : un objet est renommé au plus log(i) fois dans une suite de i fusions. On obtient alors un temps (amorti) par opération O(logn).

12.1.3 Une astuce de plus: temps (pire cas) O(logn)

On utilise une nouvelle structure de données: un arbre où les pointeurs sont orientés vers la racine: la racine contient le nom de l'ensemble. Dans ce cas, il faut toujours fusionner selon les deux tailles : le plus grand des deux ensembles devient père de l'autre. Il n'est pas nécessaire de chercher à mettre à jour les noms (en fait ne plus les stocker).

Deux remarques :

- Pour trouver le nom d'un objet, il suffit alors de suivre les pointeurs à la racine : en montant vers la racine pour trouver le nom.
- La taille des ensembles double (au moins) à chaque itération. Donc, au plus, on a logn itérations.

12.1.4 Et encore une astuce

Chaque fois qu'on a parcouru le chemin d'un objet vers la racine, on le parcourt une deuxième fois, en remplaçant chaque pointeur par un raccourci directement à la racine !

Remarque : Le pire des cas d'une opération reste O(logn). Mais chaque recherche rend plus rapide des recherches à venir. Dans ce cas, quel est l'effet sur le temps total ?

12.1.5 Les fonctions $\log^*$ et T

$\text{Log}^*(n)$ est une fonction qui croît très lentement avec n. Le nombre de fois qu'il faut appliquer $\log_2$ à n (avec arrondi vers le bas chaque fois) pour arriver à 1. T(n) une fonction qui croît TRES vite avec n : la valeur d'un tour de n fois 2. $\log^*$ est l'inverse de T.

On va démontrer que le temps (amorti) d'une opération de fusion/recherche est $O(\log^*(n))$.

12.2 MAJORATION

12.2.1 Majorer le temps amorti: (1) les gros sauts de taille

Le temps d'une fusion est constante : on ne fait qu'ajouter un pointeur et sommer les deux tailles. Le temps de suivre un pointeur en cherchant la racine est « facturé » à l'opération de recherche ou à l'opération de fusion

qui a ajouté l'objet comme un fils; selon les tailles des ensembles père et fils. Si les deux tailles ont des valeurs différentes de $\log^*$, facturer à la recherche. Le nombre facturés à une recherche est forcément $O(\log^*(n))$.

12.2.2 Majorer le temps amorti: (2) les petits sauts

Si les deux tailles ont la même valeur de $\log^*$, facturer à la fusion. Pour un ensemble de taille s, où $T(i) \leq s \leq T(i+1)$, le nombre facturé est au plus $\log_2 T(i+1)$, c'est-à-dire $T(i)$ et donc $\leq s$. Deux sous-arbres sont disjoints si le rapport entre leurs deux tailles (grand:petit) est inférieur à 2. Donc :

- pour les s tels que $T(i) \leq s < 2T(i)$, le total facturé est inférieur à n
- pour ceux tels que $2T(i) \leq s < 4T(i)$, à n/2.
- pour les s tels que $T(i) \leq s < T(i+1)$, à 2n
- et enfin, pour tous les s, inférieur à $2n\log^*(n)$

12.2.3 Conclusion

On obtient donc :

- temps total O((nombre de fusions + nombre de recherches)*$\log^*(n)$)
- temps amorti de chaque opération $O(\log^*(n))$

Il n'est pas évident que ce majorant soit strict (pour cet algorithme ou un autre).

13 RECHERCHE D'OCCURRENCES D'UNE CHAINE DE CARACTERES DANS UNE AUTRE

Si l'on considère une chaîne longue le texte dans un tableau T[1..n] et une chaîne courte le cible dans un tableau C[1..m] ; vérifier qu'une occurrence de C commence à T[i] prend un temps m. Donc, un algorithme simple en temps se retrouve avec la complexité O(mn).

Comment faire mieux ?

13.1 L'ALGORITHME DE KNUTH-MORRIS-PRATT

On lit le tableau de texte T de gauche à droite. A chaque moment, on stocke la (taille de la) partie au début de C qui correspond à la partie de T qu'on vient de lire. Cette taille peut être mise à jour sans retour en arrière.

Il faut savoir, pour chaque préfixe de C, quel est son plus long suffixe qui est aussi un préfixe de C.

13.1.1 Un automate qui effectue la recherche du suffixe-préfixe

On créé un état pour chaque lettre de C, mettons par exemple $C=C_1,...,C_m$. Considérons un état C_i : on a déjà lu $C_1,...,C_i$. On continue d'avancer si la prochaine lettre x est la bonne C_{i+1} (sauf si on a trouvé un C complet). Sinon on recule. Problème : De combien doit-on reculer lorsqu'on est à C_j ; où $C_1,...,C_j$ est le plus long préfixe de C qui est aussi suffixe de $C_1,...,C_i$ x.

13.1.2 Le pré-calcul

Le pré-calcul (du plus long préfixe de C qui est aussi suffixe de $C_1,...,C_i$) peut se faire facilement en temps $O(m^2)$ mais ce n'est pas optimal :

- Une récurrence pour S_i, la longueur de ce préfixe-suffixe si $C_{i+1} = C_{Si+1}$, $S_{i+1} = S_i+1$, sinon, si $C_{i+1}=C_{SSi+1}$, $S_{i+1}=S_{Si}+1$, etc.
- Le calcul par programmation dynamique
- Le temps du pré-calcul : O(m)
- Le temps total O(m+n)

13.2 L'ALGORITHME DE BOYER ET MOORE: L'IDEE

Comment faire mieux qu'un algorithme qui considère chaque lettre une seule fois? C'est impossible (dans certains cas au moins). Mais souvent, si C est de taille moyenne, on peut exclure des sections de T sans les regarder !

exemple trivial: si C est « aaaaaaaaaa », et T ne contient pas de a, on peut trouver la solution en ne regardant que chaque 10-ème lettre de T.

13.2.1 L'algorithme de Boyer et Moore: un peu de détail

Pour trouver les positions où une occurrence de C pourrait terminer, il faut :

- Regarder d'abord la lettre T[m]
- Si cette lettre n'a pas d'occurrences en C, avancer par m
- Si sa dernière occurrence en C est à position j (j<m), avancer par m-j
- Si sa dernière occurrence est à C[m], chercher une occurrence complète (de droite à gauche)

13.2.2 Récupération après un échec

La récupération après un échec et le redémarrage après une réussite quand on veut trouver toutes les occurrences peut se faire de la manière suivante :

- Système pareil à celui de Knuth-Morris-Pratt d'utiliser l'information obtenue pendant la partie réussie de la recherche.
- Donne temps « moyen » O(n/m) si on accepte l'idée que les deux chaînes sont aléatoires.

14 ALLOCATION ET LIBERATION D'ESPACE

Une allocation d'espace est nécessaire pour toute structure qui ne respecte pas les règles des variables locales des fonctions. De la même manière, *une libération est aussi nécessaire lorsque la variable n'est plus utilisée*, sinon une longue exécution du programme risque d'échouer, faute d'espace mémoire disponible .

Attention, libérer trop ou trop tôt une variable peut aussi être désastreux. La gestion de la mémoire peut être automatique, semi-automatique ou artisanale selon le langage de programmation et l'application.

14.1 METHODES DE LIBERATION AUTOMATIQUES

Deux types de méthodes ont été étudiés qui permettent de récupérer automatiquement l'espace qui ne sera plus utilisé par le programme. Un programme peut aussi utiliser les mêmes méthodes de « ramassage de miettes ».

Une première méthode consiste à compter les pointeurs :

- Dans chaque bloc (structure, etc.), on maintient un champ caché qui est le nombre de pointeurs au bloc,
- On met à jour ce champ à chaque fois qu'un pointeur est ajouté ou enlevé.
- Quand le nombre tombe à 0, le bloc peut être libéré (car il n'est plus pointé, donc utilisé) avec éventuellement d'autres libérations (réaction en chaîne).
- Cette méthode ne réussit pas à libérer toutes les structures circulaires (le cas simple d'un pointeur d'un bloc vers lui-même peut être traité).

14.1.1 Libération automatique marquer et balayer

Les structures accessibles à un moment donné sont celles directement connues du programme (valeurs de variables et paramètres) ainsi que celles accessibles indirectement de celles-ci par des chaînes de pointeurs. Une méthode de libération consiste à :

- maintenir un bit dans chaque bloc : est-il accessible ou non?
- de temps en temps remettre tous les bits à 0 ;
- procéder récursivement à partir de chaque bloc directement accessible en mettant des bits à 1 à chaque bloc rencontré (et arrêtant quand un bloc trouvé est déjà marqué)

- enfin, balayer toute la zone en récupérant tous les blocs non marqués (éventuellement en compactant tous les blocs survivants, mais ceci nécessite une modification de tous les pointeurs!). C'est ce qu'on appelle le ramassage.

Remarque : Ici, un problème se pose : le programme doit s'arrêter pendant ce temps de mise à jour !

14.1.2 Ramassage parallèle

Une famille d'algorithmes basés sur un premier algorithem de Dijkstra permettent de ramasser sans arrêter les autres processus. Il s'agit en général d'algorithmes compliqués dont l'efficacité a été prouvée avec plusieurs démonstrations (dont certaines sont fausses). Le principe de fonctionnement de ce type d'algorithme peut être résumé ainsi :

- On distingue trois types de bloc : des blocs non marqués (blancs), des blocs marqués mais dont les fils ne sont pas forcément marqués (gris) et des blocs marqués ainsi que ses fils (noirs)
- Quand un pointeur dans un bloc change, le bloc et le nouveau fils deviennent gris.
- C'est la méthode (garbage collector) utilisée en java.

14.1.3 Contraintes sur le langage de programmation pour qu'un système automatique soit possible

Il faut que le système puisse savoir où sont les pointeurs dans chaque bloc. On doit donc utiliser des règles strictes de typage et faire attention aux dangers d'arithmétique sur les pointeurs. De plus, il est nécessaire d'avoir des restrictions sur les types **union**.

14.2 SYSTEME SEMI-AUTOMATIQUE

Un système semi-automatique peut être par exemple implémenté en C suivant l'algorithme en exalgo suivant :

- Libération explicite par le programme (free() ou malloc() en C)
- Parcours de listes ou d'arbres pour libérer chaque nœud inutilisé (mais cela reste difficile avec structures circulaires)
- Danger subtil de parcours simple qui marque et libère!
- Attention aussi aux structures partagées (globales)

14.2.1 Allocation « artisanale » avec blocs de la même taille

Si un programme n'utilise que des blocs d'une même taille (manipule un type de liste ou d'arbre, ...), il est facile de garder tous les blocs disponibles dans une liste. Dans ce cas :

- L'allocation consiste à prendre le premier bloc de la liste (s'il y en a) ;
- La libération : ajouter un élément au début de la liste.

14.2.2 Blocs de tailles différentes

Avec des blocs de taille différente, la tâche est plus compliqué : on a toujours une liste de blocs disponibles mais de tailles différentes et en ordre d'adresse. Dans ce cas :

- L'allocation consiste prendre le premier (?) bloc suffisamment grand dans la liste, voire enlever d'autres éléments de la liste, ou les déplacer afin d'ajuster la taille nécessaire à la nouvelle allocation.
- La libération consiste à trouver la bonne position dans la liste.
- insérer et éventuellement fusionner avec son prédécesseur et/ou successeur

Remarque : le compactage est possible mais complexe !