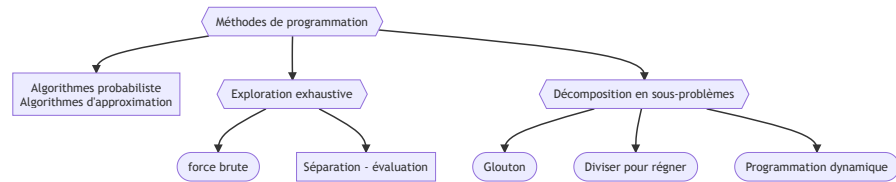


Diviser pour régner

Introduction



I. Principe

Résolution d'un problème en 3 étapes

1. *Diviser* le problème P en sous problèmes \rightarrow apparition d'un certain nombre de sous-problèmes de même instance que P
2. *Régner*: résoudre les sous problèmes de manière récursive ou non
3. *Combiner* les solutions au sous problèmes en une solution à P

a. Diviser

Division d'un problème P en sous problèmes de taille environ $\frac{n}{b}$.

Par exemple,

- On divise en 2 sous problèmes de taille identique $b = 2$
- On divise en 2 sous problèmes de taille inégale
- Autres combinaisons de sous problèmes

On divise les problème jusqu'à ce que les sous problèmes aient une résolution triviale.

Lien avec les algorithmes récursifs:

- un problème non trivial : appel récursif
- un problème trivial : cas de base

Problème lié avec la récursivité: taille de la pile. Solution: arrêt des appels récursifs quand un seuil s est atteint. Ce seuil s peut dépendre de:

- des ressources machines
- type de données
- \vdots

Il est possible de combiner les méthodes de programmation pour gagner en efficacité. Par exemple, pour trouver des racines à une fonction.

Il est possible d'avoir plusieurs cas de base:

- moins efficace du point de vue temporel
- plus efficace du point de vue gestion de la mémoire

Le coût engendré par l'étape de division est noté $D(n)$

b. Régner

Obtention d'un ou de plusieurs sous problème(s) trivial(aux) : résolution directe (cas de base)

Possibilité d'obtenir des sous problèmes un peu différents du problème : intégration de ces cas dans l'étape combiner

On note le coût engendré par l'étape régner $aT\left(\frac{n}{b}\right)$

Par exemple,

- pour une division en 2 sous problèmes de taille égale dont les 2 doivent être résolus, $a = 2$, $b = 2$. On a donc $2T\left(\frac{n}{2}\right)$
- pour une division en 2 sous problèmes de taille égale dont un seul doit être résolu, $a = 1$, $b = 2$. On a donc $T\left(\frac{n}{2}\right)$

c. Combiner

Combinaison des solutions au sous problèmes (même ou différente instance)

Production de la solution au problème P

Coût: $C(n)$

Coût temporel:

Forme général:

$$T(n) = aT\left(\frac{n}{b}\right) + D(n) + C(n)$$

Remarque: les parties entières $\lfloor \cdot \rfloor$ sont généralement omises. Par exemple, $\frac{n}{2}$ si n est impaire

Conditions souvent ignorés: $n = 1$ ou $n = s$ avec un coût supposé en $\Theta(n)$. Ce n'est pas forcément le cas (non traité)

II. Exemples déjà vus en cours

- Recherche de valeur dans un tableau trié
 $\Theta(n)$ pour la version itérative
 $\Theta(\ln n)$ pour la version diviser pour régner (dichotomie) On a $a = 1$ et $b = 2$ car le tableau est divisé en 2.
- Exponentiation a^n
 $\Theta(n)$: coût linéaire pour la version itérative
 $O(\ln n)$ pour la version récursive car le problème est divisé par 2 à chaque appel récursif

III. Autres exemples

Algorithmes de tri

Stratégie:

- exploration exhaustive (toutes les possibilités): bogosort
- glouton: tri par insertion
- diviser pour régner: diviser en sous tableaux dont le tri est trivial (tri fusion)

Déjà vus:

- tri par insertion (glouton exact) meilleur: $O(n)$, pire: $O(n^2)$
- tri par sélection (glouton) $O(n^2)$

Tri fusion:

Complexité: $O(n \ln(n))$ car

- Division: pour une profondeur de k , on a $2^k \geq n \geq 2^{k+1}$ donc $n = e^{k \ln 2}$ et donc $k = \log_2(n)$
- Combinaison: On a n comparaisons car il y a n éléments dans le tableau

Mise en place :

Fonction TriFusion(T, inf, sup)

Si inf = sup

Retourner T

Sinon

m <- floor((inf + sup) / 2)

TriFusion(T, inf, m)

TriFusion(T, m+1, sup)

```

        Fusion(T, inf, m, sup)
    Fin Si
Fin

Fonction Fusion(T, inf, m, sup)
    taille1 <- floor(m - inf) + 1
    taille2 <- floor(sup - m)

    gauche <- []
    droite <- []

    Pour i = inf à sup
        Si i <= m
            gauche[i] <- T[i]
        Sinon
            droite[i - m] <- T[i]
        Fin Si
    Fin Pour

    i <- 0
    j <- 0
    k <- 0

    Tant que i < taille1 && j < taille 2
        Si gauche[i] < droite[j]
            T[k] = gauche[i]
            i <- i + 1
        Sinon
            T[k] = droite[j]
            j <- j + 1
        Fin Si

        k <- k + 1
    Fin tant que

    Si i > taille1
        Tant que j > taille2
            T[k] <- droite[j]
            k <- k + 1
            j <- j + 1
        Fin tant que
    Sinon
        Tant que i > taille1
            T[k] <- gauche[i]
            k <- k + 1
            i <- i + 1

```

```

        Fin tant que
    Fin Si
Fin

```

Tri rapide

Principe:

- positionner une valeur directement à sa place définitive
- placer les plus petits à gauche et les plus grands à droite

Avantage: économe en place (pas de nouveau tableau, pas de variable auxiliaire)

La fonction `TriRapide` sépare le tableau en fonction de la position de la clé

```

Fonction TriRapide(T, inf, sup)
    Si sup <= inf
        Retourner T
    Fin Si

    clé <- Position(T, inf, sup)

    TriRapide(T, inf, clé)
    TriRapide(T, clé + 1, sup)
Fin

```

```

Fonction Position(T, inf, sup)
    clé <- T[inf]
    i <- inf // gauche
    j <- sup // droite

    Tant que j >= i
        Tant que i < j et T[i] < clé
            i <- i + 1
        Fin Tant que

        Tant que i < j et T[j] > clé
            j <- j - 1
        Fin Tant que

        Si j > i et T[i] > T[j]
            Échanger(T, i, j)
        Fin Si
    Fin Tant que

    Échanger(T, 0, j)

    Retourner j

```

Fin

```
Fonction Échanger(T, i, j)
    temp <- T[j]
    T[j] <- T[i]
    T[i] <- temp
```

Fin

Étude des boucles

- boucle sur i
terminaison: $j - i + 1$
invariant: $\forall k \in \llbracket \text{inf} + 1, i \rrbracket, T[k - 1] \leq \text{cle}$
- boucle sur j
terminaison: j
invariant: $\forall k \in \llbracket j, \text{sup} \rrbracket, T[k + 1] \geq \text{cle}$
- boucle extérieur:
terminaison: $j - i + 1$
invariant: combinaison des invariants des deux boucles

Complexité:

- meilleur des cas: $O(n \ln(n))$
- pire des cas: $O(n^2)$

Autre algorithme:

```
Fonction Position(T, inf, sup)
    cle <- T[inf]
    j <- sup + 1

    Pour i = sup à inf + 1 (décroissant)
        Si T[i] >= cle
            j <- j - 1
            Échanger(T, i, j)
        Fin Si
    Fin Pour

    Échanger(T, inf, j-1)

    Retourner j-1
```

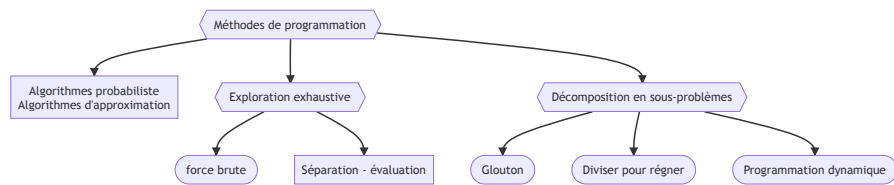
Fin

Programmation dynamique

Objectifs du cours:

- Cadre pour les techniques de programmation
- Programmation dynamique

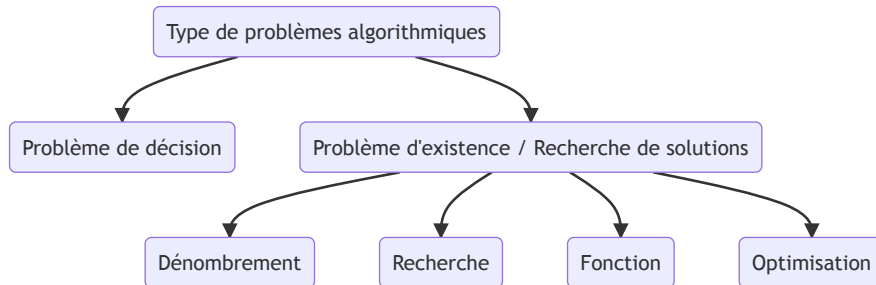
I. Introduction - Cadre pour l'algorithmique



Formulation d'un problème en algorithmique:

- Soit P un problème quelconque
- Soit I une instance de P
- Soit A un algorithme résolvant le problème P

Quelles sont les familles de problèmes P ?

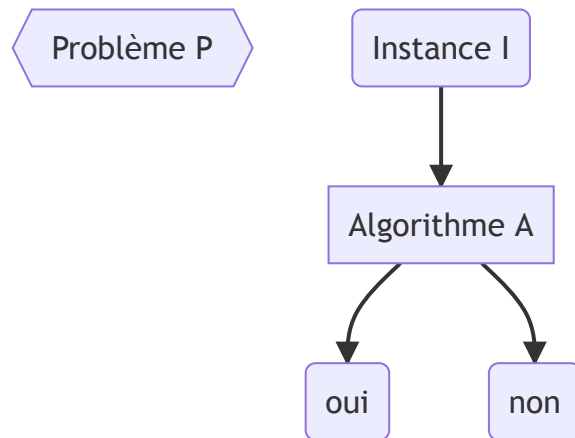


Problème de décision (ou reconnaissance)

Objectif : validation de l'existence d'une solution

A chaque instance du problème P , la réponse apportée par l'algorithme est oui ou non (résultat binaire)

Problème de classification



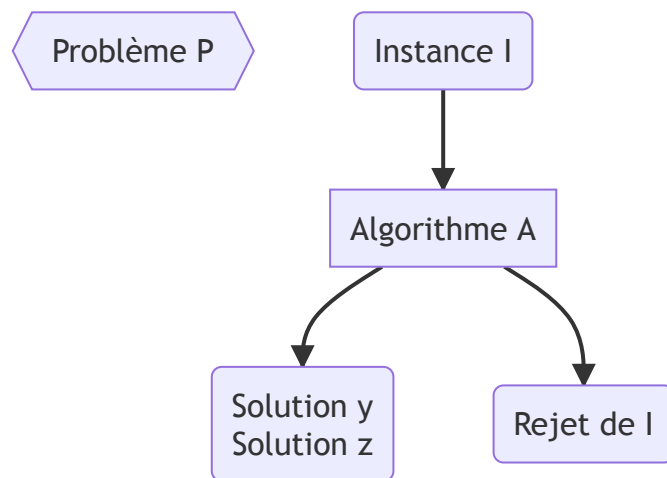
Exemple: soit n un entier naturel, déterminer si n est un nombre premier

Problème de recherche

Objectif: trouver une solution lorsqu'un algorithme n'est pas écrit mais que la spécification de la solution est connue.

Trouver, si elle existe, une solution associée à une instance d'entrée grâce à une relation figurant dans l'énoncé de la recherche.

Problème de décision : restriction d'un problème de recherche



Exemple : trouver les facteurs premiers d'un entier naturel n

Problème de dénombrement

Objectif : dénombrer les solutions

Déterminer le nombre de solutions à un problème de recherche

Sous problème du problème de dénombrement et de décision :

- unicité d'une solution optimale
- énumération des solutions

Exemple: soit n un entier naturel, déterminer le nombre de facteurs premiers non triviaux de n

Problème d'optimisation

Objectif : recherche de la meilleure solution parmi toutes les solutions possibles d'un problème P sur une instance I

Détermination d'un jeu de paramètres en entrée d'une fonction donnant à cette fonction la valeur maximale ou minimale

Optimisation sous contrainte ou sans contrainte (méthode de descente, programmation linéaire, ...)

Exemple: recherche du plus grand facteur premier non trivial d'un entier naturel n

Problème de fonction

Objectif : engendrer les solution

Un seul résultat attendu pour chaque entrée

Plus complexe que le problème de décision car le résultat est une valeur plutôt qu'un résultat binaire

Exemple: problème de régression \rightarrow prévision de valeur

II. Programmation dynamique: exemple

Découpe de barres d'acier

Entreprise \rightarrow achat et revente de barres d'acier

Coupe de barres d'acier pour optimiser les profits

Hypothèses:

- Découpe d'une barre en barres de $i = 1, 2, \dots$ cm
- Prix des tronçons de i cm à la revente connus :

Longueur i (cm)	1	2	3	4	5	6	7	8	9	10
Prix	1	5	8	9	10	17	17	20	24	30

Bilan des présentations:

- Pas facile de présenter un code \implies algorithme ?
- Attention à la lisibilité (couleur et taille)
- Trop de code \rightarrow difficile à appréhender
- Position / pertinence des commentaires
- Pertinence des éléments sur les diapos
 - Utilité
 - Aide / illustration du discours
- Définir un plan
- Exemples
- Éviter trop de textes / phrases trop longues
- Travail en équipe \rightarrow répartition des rôles
 - oral
 - élaboration du diapo
 - relecture des diapos

Présentation orale:

1. Contexte / Présentation du problème
2. Plan
3. Méthodes utilisées
 - Formulation mathématique du problème (éventuellement)
 - Choix / justification du choix d'une méthode de programmation
4. Détail / Mise en œuvre des méthodes
 - Architecture global ? (liste des fonctions, et rôles de ces fonctions)
 - Algorithmes pour les principes (à dérouler ?)
 - Code pour les particularités de traduction \rightarrow exploitation des langages
 - Correction / terminaison des algorithmes
5. Bilan / Comparaison des performances ? \rightarrow complexité

Le revenu maximum est donc $r_n = \max(p_n, p_1 + r_{n-1}, \dots, p_{n-1} + r_1)$.

Revenu maximum : $r_n = \max_{i \in \mathbb{N}}(p_i + r_{n-i})$

La valeur de i qui maximise r_n est inconnue: il faut tester toutes les valeurs

Décomposition du problème initial en 2 sous problèmes du même type de taille plus petite, indépendants \rightarrow instances plus petites du problème initial

Solution optimale global: celle qui maximise chacun des 2 sous problèmes

Autres manière d'organiser la structure:

- Structure récursive : découpe = tronçon de gauche de longueur i ne pouvant être découpé + tronçon pouvant être découpé de longueur $n - i$
- Solution sans coupe
 - Le tronçon de coupe est de taille $i = n$, de revenu p_n
 - le revenu du tronçon de droite restant de taille 0 est $r_0 = 0$

Nouvelle expression de l'optimisation de la coupe :

$$r_n = \max_{0 \leq i \leq n} (p_i + r_{n-i})$$

Implémentation utilisant la récursivité (cf. présentations) \rightarrow complexité temporelle : $O(2^n)$

Optimisation de la récursivité par la programmation dynamique

Fonction récursive inefficace: plusieurs évaluations identiques \rightarrow chevauchements des sous problèmes

Idée: mémoriser les résultats pour ne les évaluer qu'une seule fois

- consulter les résultats mémorisés
- Si évaluation déjà produite, la renvoyer
- Sinon, procéder à l'évaluation

Compromis temps/mémoire

- Programmation dynamique plus coûteuse en mémoire mais gain temporel
- Recherche d'une complexité polynomiale plutôt qu'exponentielle
 - Coût polynomial si chacun des sous problèmes distincts se résout en temps polynomial par rapport à la taille des données d'entrée : $O(n^\alpha)$.

Idée: mémoriser les résultats pour ne les évaluer qu'une seule fois

- approche descendante : mémoïsation
- approche ascendante

Approche descendante

- Écriture de la fonction récursive
- Modification
 - Introduction d'une structure permettant de sauvegarder les évaluations

- structure utilisée : tableau ou table de hachage
- Vérification si le sous problème est déjà évaluée avant les appels récursifs

Approche ascendante:

- Associée à une notion de taille dans les sous problèmes
- Tri des sous problèmes par taille
- Résolution des sous problèmes en commençant par les plus petits

Fonction CoupeBarreMemo(p,n):

Entrée: p, le tableau des revenus et n taille de la barre (entier)

Sortie: q, revenu maximum (entier)

Début

memo = CreerTableau(dimension: n, valeurs: -1)

Fonction Aux(p, n, memo):

Début

Si memo[n] >= 0 alors

Retourner memo[n]

Fin Si

Si n = 0 alors

q = 0

Sinon

q = -1

Pour i allant de 1 à n faire

q = max(q, p[i] + Aux(p, n-i, memo))

Fin Pour

Fin Si

memo[n] = q

Retourner q

Fin

Retourner Aux(p, n, memo)

Fin

Ordre des appels récursifs pour $n = 4$:

$4 \rightarrow 3 \rightarrow 2 \rightarrow 1$

donc programmation dynamique *descendante*

Fonction CoupeBarreAsc(p, n)

Entrée: p, tableau des revenus et n, taille de la barre (entier)

Sortie: q, revenu maximum (entier)

```

Début
  asc = CreerTableau(dimension : n + 1, valeurs : -1)
  asc[1] = 0

  Pour j allant de 2 à n + 1 faire
    q = -1

    Pour i allant de 1 à j - 1 faire
      q = max(q, p[i] + asc[j-i])
    Fin Pour

    asc[j] = q
  Fin Pour

  Retourner asc[n]
Fin

```

III. Principe de la programmation dynamique

Historique

Programmation dynamique → planification de tâches (Richard Bellman 1953)

Résolution de problèmes d'optimisation

Cadre d'utilisation

Chevauchement de sous problèmes: évaluation redondante (plusieurs appels d'une fonction avec les mêmes arguments)

Ordre total → possibilité d'ordonner les sous problèmes

La solution d'un problème se calcule à partir des solutions des sous problèmes

→ Optimisation d'une programmation récursive

Conception

Définition des sous problèmes

Identifier la relation de récurrence entre les sous problèmes

Nécessité de comprendre les interdépendances entre les sous problèmes

En déduire un algorithme récursif avec approche ascendante et descendante

Méthode ascendante et descendante

Résoudre le sous problème original à partir des solutions des sous problèmes

Nécessité de comprendre les interdépendances entre les sous problèmes

Arbre des appels récursifs → identification des chevauchements

Mise en place du graphe des sous problèmes:

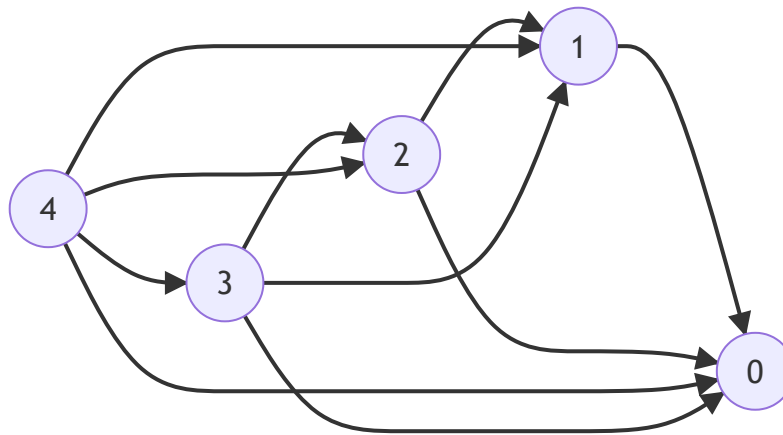
Graph orienté

- sommet pour chaque sous problème
 - arc orienté allant du sous problème x au sous problème y
 - solution optimale de x dépend de la solution optimale de y

Taille du graph donne une indication sur le temps d'exécution

Chaque sous problème n'est résolu qu'une fois → somme des temps

Exemple: découpe de barre avec $n = 4$



Méthode ascendante:

Résolution des sous problèmes les plus petits en premier

Nécessité d'un ordre total sur l'ensemble des sous problèmes → sous problème traité lorsque les sous problèmes dont il dépend ont été résolus

Méthode descendante (mémoïsation):

Mémoïser: Conserver les résultats à la fin de l'exécution d'une fonction le résultat associé aux arguments d'appels pour éviter d'avoir à recalculer

Parcours en profondeur du graph de sous problèmes

Complexité

Temporelle \rightarrow polynomiale n^α

Nombre de sous problèmes à évaluer

Temps nécessaire pour évaluer chaque sous problème

nombre de sous problèmes \times complexité par sous problèmes (sans compter les appels récursifs)

Spatiale

Nombre de sous problèmes dont il faut garder la trace à chaque étape

Bilan

Exploration exhaustive \rightarrow Force brute

Détermination de toutes les solutions

Choix d'une des solutions

Décomposition en sous problèmes

- Diviser pour régner
 - Décomposer en sous problèmes
 - Instances plus petites du problème initial
 - Obtention de sous problèmes triviaux
 - Combinaison des solutions triviales
- Gloutons
 - Choix d'une solution optimale à un problème élémentaire
 - Incorporation progressive des solutions
- Programmation dynamique
 - Décomposer en sous problèmes
 - Instances plus petites du problème initial
 - Chevauchement des sous problèmes
 - Ordre total parmi les sous problèmes
 - Mémorisation des résolutions

Algorithmes gloutons

c.f. polycopié

II. Exemple : choix d'activités

Sous-structure optimale :

Ajout de a_k dans $A_{ij} \rightarrow$ définition de deux sous-problèmes dans S_{ij} .

On peut définir

$$A_{jk} = A_{ij} \cap S_{ik}$$

$$A_{kj} = A_{ij} \cap S_{kj}$$

On a donc $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ avec A_{ik} avec $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$.

À démontrer : la solution optimale A_{ij} doit-elle nécessairement inclure les solutions optimales de S_{ik} et S_{kj} ?

Hypothèse : On peut bâtir A'_{ik} , sous ensemble d'activités mutuellement compatibles dans S_{ik} tel que $|A'_{ik}| > |A_{ik}|$.

Alors, on peut trouver un sous-ensemble A'_{ij} tel que $A'_{ij} = A'_{ik} \cup \{a_k\} \cup A_{kj}$. On a alors $|A'_{ij}| = |A'_{ik}| + |A_{kj}| + 1 > |A_{ij}| = |A_{ik}| + |A_{kj}| + 1$.

Ce qui contredit l'hypothèse initiale affirmant que A_{ij} est un sous-ensemble de plus grande taille d'activités mutuellement compatible de S_{ij} .

Raisonnement similaire pour A_{kj} .

Conclusion: La solution optimale de S_{ij} inclut les solutions optimales de S_{ik} et S_{kj} .

Chevauchements de sous-problèmes :

Taille d'une solution optimale pour S_{ij} contenue dans le tableau à 2 dimensions : $|A_{ij}| = c[i, j]$.

Soit $c[i, j] = c[i, k] + c[k, j] + 1$.

En pratique, on ignore si la solution optimale du sous-problème S_{ij} contient la solution a_k .

Nouvelle formulation : examen de toutes les activités de S_{ij} pour déterminer k :

$$c[i, j] = \begin{cases} 0 & \text{si } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{si } S_{ij} \neq \emptyset. \end{cases}$$

D'où les chevauchements de sous-problèmes (résoudre $S_{i,m+1}$ nécessite de résoudre S_{im}).

Programmation dynamique envisageable mais lourde à mettre en œuvre.

c.f. polycopié

Est-ce que le choix glouton fait toujours partie de la solution optimale ?

Théorème:

Soit un sous-problème S_k non vide. Soit a_m l'activité de S_k ayant l'heure d'achèvement la plus précoce.

Alors, a_m est incluse dans un certain sous-ensemble de taille maximale d'activités mutuellement compatibles de S_k .

Démonstration:

Soit A_k un sous-ensemble de taille maximale d'activités mutuellement compatibles de S_k . Soit a_j la première activité de A_k .

Si $a_j = a_m$, alors le théorème est démontré.

Si $a_j \neq a_m$, soit $A'_k = (A_k \setminus \{a_j\}) \cup \{a_m\}$, a_m ayant l'heure d'achèvement la plus précoce dans S_k : $f_m \leq f_j$. A'_k est aussi un sous-ensemble d'activités mutuellement compatibles de S_k . Par ailleurs, $|A'_k| = |A_k|$, donc A'_k est un sous-ensemble de taille maximale d'activités mutuellement compatibles de S_k , contenant a_m .

Le théorème est donc démontré.

Mise en œuvre des algorithmes :

- Récursif:

Fonction ChoixActivitesRecurisif(d, f, k, n)

Entrée: d, tableau des dates de début des activités;
f, tableau des dates de fin des activités;
k, indice de l'activité courante (entier);
n, longueur du tableau d et f (entier).

Sortie: A, combinaison d'activités.

Précondition: f et d sont triés par date de fin croissantes.

Post-condition: A est un sous-ensemble de taille maximale d'activités mutuellement compatibles.

Début

m := k + 1

Tant que m <= n et d[m] <= f[k] faire

m := m + 1

Fin Tant que

Si m <= n alors

```

    Retourner Combiner({ m }, ChoixActivitesRecuratif(d, f, m, n))
Sinon
    Retourner  $\emptyset$ 
Fin Si
Fin

Complexité en  $\Theta(n)$ 

```

- Itératif:

```

Fonction ChoixActivitesIteratif(d, f, k, n)
    Entrée: d, tableau des dates de début des activités;
           f, tableau des dates de fin des activités.
    Sortie: A, combinaison d'activités.
    Précondition: f et d sont triés par date de fin croissantes.
    Post-condition: A est un sous-ensemble de taille maximale
                   d'activités mutuellement compatibles.

    Début
        n := d.longueur
        A := { 1 }
        k := 1

        Pour m allant de 2 à n faire
            Si d[m] >= f[k] alors
                A := Combiner(A, { m })
                k := m
            Fin Si
        Fin Pour

        Retourner A
    Fin

    Complexité en  $\Theta(n)$ 

```

Bilan

c.f. polycopié

1. Détermination de la sous-structure optimale :

$$A_{ij} = A_{ik} + \{a_k\} + A_{kj}$$

2. Formulation de la récurrence (mais non exploité pour un algo. récursif)

$$c[i, j] = \begin{cases} 0 & \text{si } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{si } S_{ij} \neq \emptyset. \end{cases}$$

3. Le choix glouton conduit à un unique sous-problème:

$$A_k = \{a_k\} + \text{solution optimale au sous-problème } S_k$$

4. Le choix glouton + théorème
5. Algorithme récursif
6. Algorithme itératif

III. Codage Huffman

Objectif : Économiser l'espace mémoire; limité aux suites de caractères pour le cours MP2I.

Compression de données: RLE

Code RLE déjà vu au premier semestre, en devoir : codage sans perte de données mais peu efficace sur les textes

Encodages au programme (sans perte de données):

- Lempel-Zil-Welch (LZW) → projet de fin d'année
- Huffman
 - algorithme glouton
 - très efficace pour les textes (entre 20% et 90% d'économie)
 - utilisation d'une table de fréquences d'apparition pour l'encodage
 - encodage de chaque caractère par une chaîne binaire
 - utilisé pour les fichiers **zip**, pour la compression **JPEG** et **PNG**.

Exemple :

Texte de 100 000 caractères utilisant uniquement 6 caractères distincts: 'a', 'b', 'c', 'd', 'e', et 'f'.

Fréquence d'apparition des différents caractères connus

Encodage de longueur fixe *v.s.* encodage de longueur variable.

Caractère	a	b	c	d	e	f
Fréquence ($\times 10^3$)	45	13	12	16	9	5
Encodage (longueur fixe)	000	001	010	011	100	101
Encodage (longueur variable)	0	101	100	111	1101	1100

Tailles des fichiers ?

- Encodage de longueur fixe : 300 000 bits
- Encodage de longueur variable : 224 000 bits

On a donc 25% d'économie avec l'encodage de longueur variable.

Exemple : 0 111 101 101 → a d b b

Pourquoi n'y a-t-il pas d'ambiguïté pour le décodage ?

Codage préfixe : aucun mot code n'est le préfixe d'un autre mot code.

Un codage préfixe peut toujours donner la compression maximale entre tous les codages de caractère.

Utilisation d'un arbre binaire :

Feuilles → caractères et fréquences d'apparition des caractères

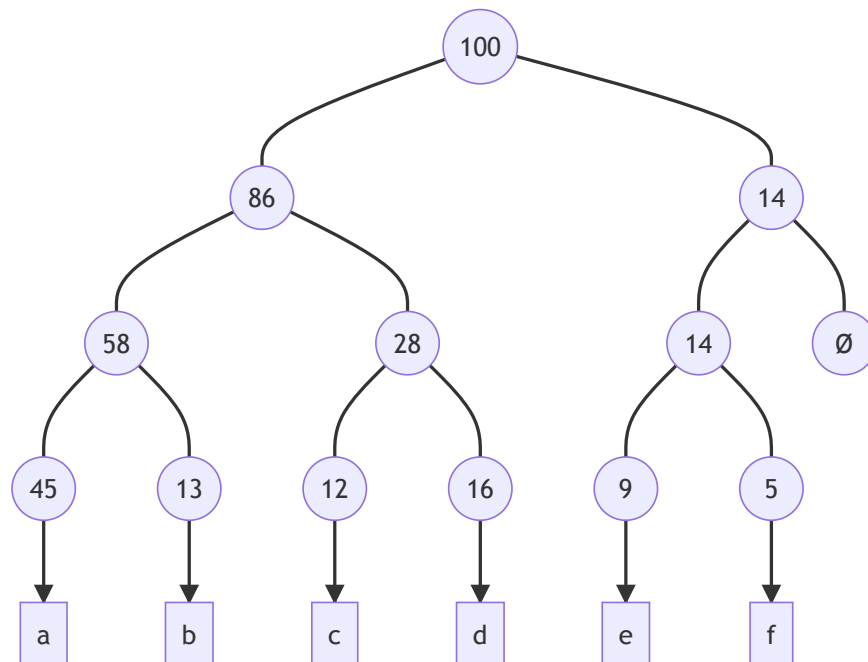
Étiquettes des nœuds internes et racines → fréquences d'apparition

Vers le fils gauche → 0

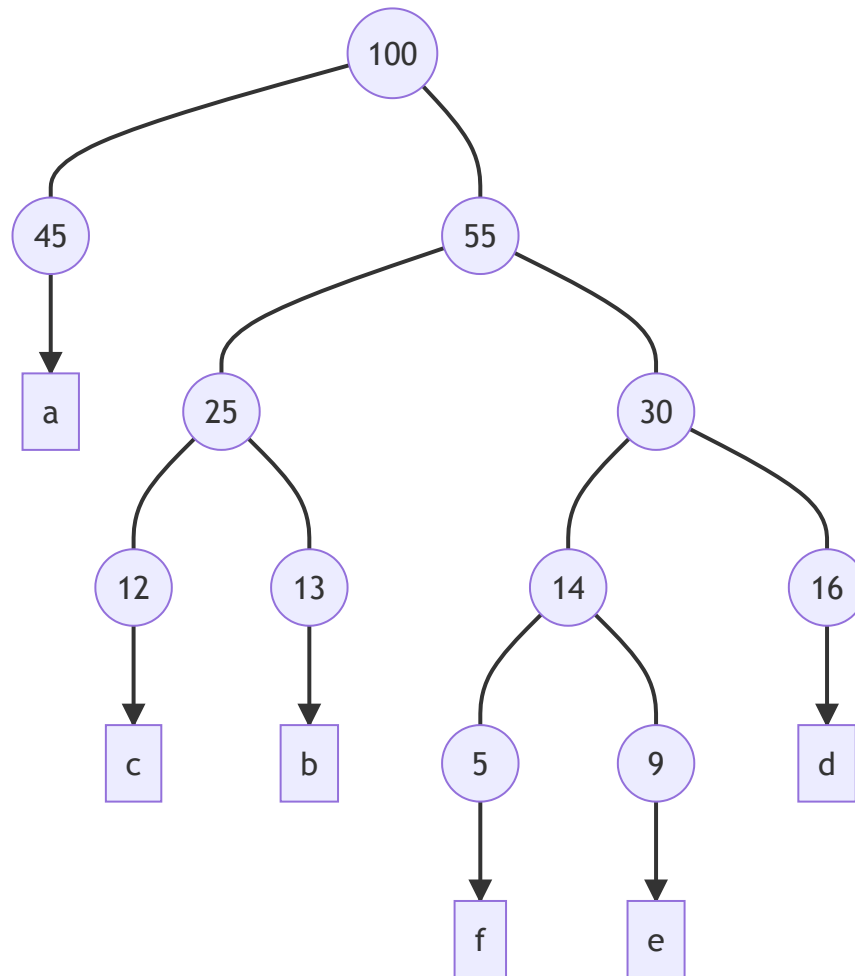
Vers le fils droit → 1

Codage optimal → arbre binaire dont les nœuds sont d'arité 2.

Encodage de longueur fixe:



Encodage de longueur variable:



Construction du codage de Huffman

Coût de l'arbre : $\sum_{i=1}^n f_i \times p_i$ où p_i est la profondeur du caractère i , f_i la fréquence du caractère i .

Utilisation d'une file de priorité min F contenant les arbres \rightarrow forêt

Pour chaque arbre de la forêt :

Chaque nœud interne contient une fréquence

Chaque feuille contient un caractère et sa fréquence d'apparition

La forêt est ordonnée par ordre de fréquences croissantes.

État initial de la forêt F_0 :

Chaque arbre ne contient qu'un seul nœud : un caractère et sa fréquence.

Donc, $\text{Card}(F_0) = n = \text{nombre de caractères différents dans le texte à encoder}$.

Construction de l'arbre :

- Extraire le minimum de la file
- Extraire le minimum de la file
- Fusionner les deux arbres extraits dans un nouvel arbre
→ structure de file min
- Insérer le nouvel arbre dans la file
- Rétablir la structure du file min

Fonction `construitHuffman(F)`

Entrée : F, file de priorité min, forêt d'arbres (files min)

Sortie : F, contenant un seul élément, l'arbre d'encodage Huffman

Pré-condition : F contient les caractères et leurs fréquences (ordre croissant)

Post-condition : F, contenant un seul élément, l'arbre d'encodage Huffman

Début

Si $\#F = 1$ alors

Returner F

Sinon

A1 = F.définir // Extraire_min(F)

A2 = F.définir // Extraire_min(F)

A = A1 + A2 // construction de l'arbre tournoi

freq[A] = freq[A1] + freq[A2] // Calcul de la fréquence de A

Insérer(A, F) // Insertion à la bonne place de A dans F

Retourner construitHuffman(F) // Appel récursif

Fin Si

Fin

Démonstration du lemme 1

a et b deux feuilles sœurs de profondeur maximale pour l'arbre T .

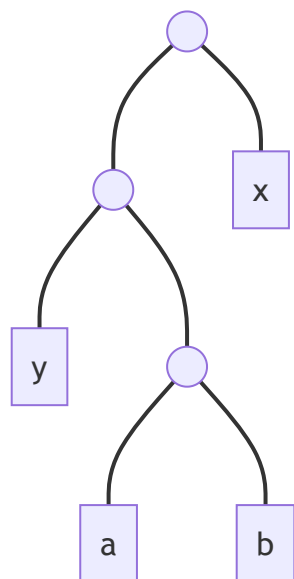
$$a.\text{freq} \leq b.\text{freq} \quad \text{et} \quad x.\text{freq} \leq y.\text{freq}$$

Comme x et y sont de fréquences les plus basses :

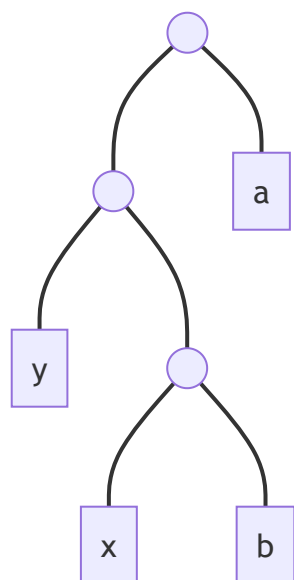
$$x.\text{freq} \leq a.\text{freq} \quad \text{et} \quad y.\text{freq} \leq b.\text{freq}$$

Si $x.\text{freq}$ alors $x.\text{freq} = a.\text{freq} = y.\text{freq} = b.\text{freq}$ donc le lemme est démontré.

Si $x.\text{freq} \neq b.\text{freq}$, on a :



En permutant les feuilles a et x , on obtient T' :



On cherche le signe de $c(T) - c(T')$.

$$\begin{aligned}
c(T) - c(T') &= x.\text{freq} \times p_x + a.\text{freq} \times p_a - x.\text{freq} \times p'_x - a.\text{freq} + p'_a \\
&= x.\text{freq} \times p_x + a.\text{freq} \times p_a - x.\text{freq} \times p_a - a.\text{freq} \times p_x \\
&= \underbrace{(a.\text{freq} - x.\text{freq})}_{\geq 0} \times \underbrace{(p_a - p_x)}_{\geq 0}
\end{aligned}$$

$$\begin{aligned}
c(T') - c(T'') &= y.\text{freq} \times p'_y + b.\text{freq} \times p'_b - y.\text{freq} \times p''_y - b.\text{freq} + p''_y \\
&= x.\text{freq} \times p'_y + b.\text{freq} \times p'_b - y.\text{freq} \times p'_b - b.\text{freq} \times p'_b \\
&= \underbrace{(b.\text{freq} - y.\text{freq})}_{\geq 0} \times \underbrace{(p'_b - p'_y)}_{\geq 0}
\end{aligned}$$

Démonstration du lemme 2

Pour chaque caractère $c \in A \setminus \{x, y\}$ on a :

$$c.\text{freq} \times p_c = c.\text{freq} \times p'_c$$

car T est bâti à partir de T' .

Pour les caractères x, y et z :

z est le parent de x et de y donc $p_x = p_y = p'_z + 1$.

D'où

$$\begin{aligned}
x.\text{freq} \times p_x + y.\text{freq} \times p_y &= (x.\text{freq} + y.\text{freq}) \times (p'_z + 1) \\
&= z.\text{freq} \times p'_z + (x.\text{freq} + y.\text{freq})
\end{aligned}$$

D'où

$$\begin{aligned}
C(T) &= C(T') \times (x.\text{freq} + y.\text{freq}) \\
C(T') &= C(T) - x.\text{freq} - y.\text{freq}
\end{aligned}$$

Raisonnement par l'absurde : T ne représente pas un codage préfixe optimal de A de coût minimal. Il existe T'' tel que $C(T'') < C(T)$

Dans T'' , x et y sont frères

Soit T'' , l'arbre T'' pour lequel le parent commun à x et y a été remplacé par z , de fréquence $z.\text{freq} = x.\text{freq} + y.\text{freq}$.

Donc $C(T'') - x.\text{freq} - y.\text{freq} < C(T) - x.\text{freq} - y.\text{freq}$.

Soit $C(T''') < C(T')$, ce qui contredit l'hypothèse affirmant que T' représente un codage préfixe optimal de coût minimal pour A' .

Donc T représente bien un codage préfixe optimal de coût minimal pour A .