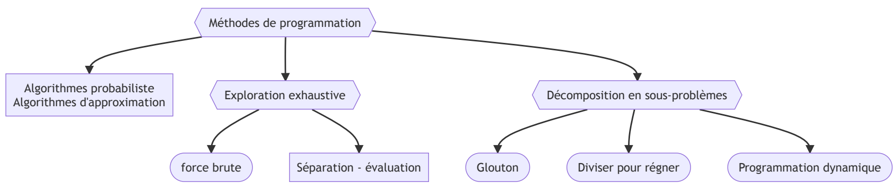
Diviser pour régner

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, $a = 2, \quad b = 2$. 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, $a = 1, \quad b = 2$. 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 $\lfloor \quad \rfloor$ 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: $O\big(n\ln(n)\big)$ car

- Division: pour une profondeur de k, on a $2^k \geqslant n \geqslant 2^{k+1}$ 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] < cle
            i <- i + 1
        Fin Tant que

        Tant que i < j et T[j] > cle
            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: $\forall k \in [\![\inf +1, i]\!], T[k-1] \leqslant cle$

- boucle sur j

  terminaison: j

  invariant: $\forall k \in [\![j, \sup]\!], T[k+1] \geqslant cle$

- boucle extérieur:

  terminaison: $j - i + 1$

  invariant: combinaison des invariants des deux boucles

Complexité:

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

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
```