Diviser pour régner

Montaigne 2023-2024

– mpi23@arrtes.net –

Recherche dichotomique

Recherche séquentielle

Étant donné un *tableau* d'objets d'un type donné, comment établir la *présence* ou l'*absence* d'un *élément* dans ce tableau ?

- Si les éléments du tableau sont *non ordonnés*, une *recherche séquentielle* répond à la question.
- Le code suivant renvoie le booléen `true` si un élément `elt` appartient à un tableau `tab`, le booléen `false` dans le cas contraire.

```
let binary_search x tab =
  let n = Array.length tab in
  let i = ref 0 in
  while !i < n && tab.(!i) != x do incr i done;
  !i < n
```

- Sa complexité temporelle au pire est $O(n)$ où n est la taille du tableau.

Recherche séquentielle

La question peut également être celle de la détermination de la *position de l'élément* recherché dans le tableau.

Si l'élément est présent dans le tableau, son indice constitue la réponse. S'il est absent, on peut choisir de renvoyer une valeur conventionnelle : `(-1)` par exemple.

```
let binary_search_pos x tab =
  let n = Array.length tab in
  let i = ref 0 in
  while !i < n && tab.(!i) != x do incr i done;
  if !i = n then i := -1;
  !i
```

La complexité temporelle au pire du code reste $O(n)$.

Recherche dichotomique

Si les éléments du tableau sont *ordonnés*, la *recherche dichotomique* est plus efficace. Elle cherche l'élément dans une moitié de tableau, cette procédure étant itérée jusqu'à trouver l'élément ou pas.

Le code suivant, vu dans le chapitre précédent, renvoie la position de `elt` dans `tab` s'il y est présent, `(-1)` sinon.

```
(* recherche de x dans tab[lo..hi[ *)
let rec binary_search_rec x tab lo hi =
  if hi <= lo then raise Not_found;
  let mid = lo + (hi - lo) / 2 in
  if x < tab.(mid)
  then binary_search_rec x tab lo mid
  else if x > tab.(mid)
    then binary_search_rec x tab (mid + 1) hi
    else mid

let binary_search x tab =
  binary_search_rec x tab 0 (Array.length tab)
```

Recherche dichotomique

La *recherche dichotomique* illustre la technique dite *diviser pour régner* qui, pour résoudre un problème, résoud un sous-problème qui contient la solution.

Dans le cas de la recherche dichotomique, cette technique peut être appliquée en raison de l'existence d'un ordre pré-existant sur les éléments du tableau.

Si le tableau n'est pas ordonné, on procède par recherche séquentielle ou on commence par ordonner le tableau (coût en $O(n \log_2 n)$ en moyenne pour un tableau de taille n).

Recherche dichotomique

La *complexité de la recherche dichotomique* peut être établie à l'aide d'une *relation de récurrence* sur les coûts des appels récursifs et les coûts des opérations annexes.

Pour un tableau de taille n, notons c_n le *coût au pire des appels récursifs et des opérations annexes* réalisés par la fonction. Dans le code précédent, les appels récursifs sont ceux de la fonction `binary_search_rec`.

On a les *cas de base* : $c_0 = 0$, $c_1 = 0$.

Dans le *cas général*, la relation est de la forme :

$$\forall n \in \mathbb{N}^* \qquad c_n = c_{\lfloor (n-1)/2 \rfloor} + \alpha \quad \text{ou} \quad c_{\lceil (n-1)/2 \rceil} + \beta$$

α et β étant des nombres positifs associés aux coûts constants des opérations annexes. Son étude, traitée plus loin dans ce cours, montre que $c_n = O(\log_2 n)$.

Exponentiation rapide

Exponentiation séquentielle

Comment calculer x^n si x et n sont des entiers naturels ?

Un premier calcul consiste à faire $n-1$ multiplications.

$$x^n = \underbrace{x \cdots x \cdots \ldots \cdots x}_{n-1 \text{ multiplications}}$$

En termes algorithmiques, cela revient à faire une boucle.

La *complexité temporelle* de ce calcul est $O(n)$ si le coût de la multiplication est constant.

```
let exp x n =
  let p = ref 1 in
  for i = 1 to n do p := !p * x done;
  !p
```

Exponentiation rapide

Un deuxième calcul s'appuie sur l'observation suivante.

$$x^n = \begin{cases} \left(x^2\right)^{n/2} & \text{si } n \text{ est pair} \\ x \cdot \left(x^2\right)^{(n-1)/2} & \text{si } n \text{ est impair} \end{cases}$$

Cet algorithme se prête naturellement à un codage récursif.

```
let rec exp x = function
  | 0 -> 1
  | n -> let y = exp (x*x) (n/2) in
         if n mod 2 = 0 then y else y*x
```

C'est encore un exemple de mise en œuvre de la technique *diviser pour régner*.

Exponentiation rapide

La *complexité de l'exponentiation rapide* peut être établie à l'aide d'une *relation de récurrence* exprimant le nombre de multiplications effectuées lors des appels récursifs.

Pour $n \in \mathbb{N}$, notons c_n ce *nombre de multiplications*.

On a le *cas de base* : $c_0 = 0$.

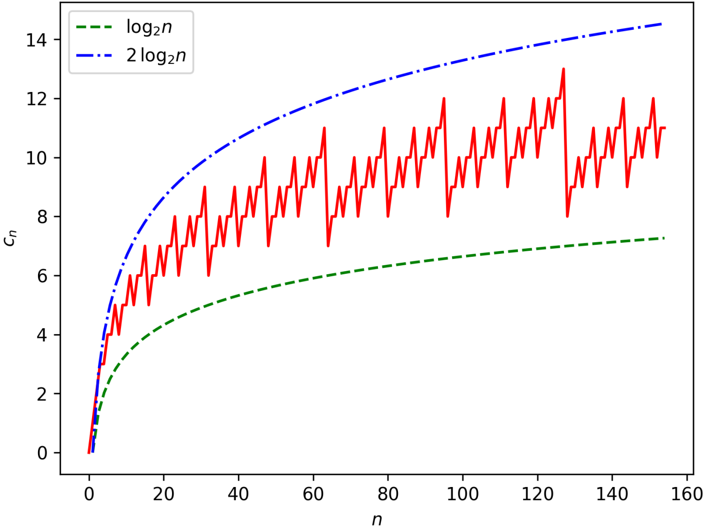
Dans le *cas général*, la relation est de la forme :

$$\forall n \in \mathbb{N}^* \qquad c_n = c_{\lfloor n/2 \rfloor} + 1 + \begin{cases} 0 & \text{si } n \text{ est pair ;} \\ 1 & \text{si } n \text{ est impair.} \end{cases}$$

Finalement $c_0 = 0$ et :

$$\forall n \in \mathbb{N}^* \qquad 1 + c_{\lfloor n/2 \rfloor} \leqslant c_n \leqslant 2 + c_{\lfloor n/2 \rfloor}$$

Exponentiation rapide



Exponentiation rapide

Étudions l'inégalité :

$$\forall n \in \mathbb{N}^* \qquad c_n \leqslant 2 + c_{\lfloor n/2 \rfloor}$$

- Soit n un entier naturel. Désignons par (u_i) la suite définie par $u_0 = n$ et, pour tout entier naturel i, $u_{i+1} = \lfloor u_i/2 \rfloor$.
- Cette suite est d'abord strictement décroisssante puis prend la valeur 1 au rang $p = \lfloor \log_2 n \rfloor$ et est nulle au-delà de ce rang.
- On peut écrire :

$$\forall i \in \mathbb{N} \qquad c_{u_i} \leqslant 2 + c_{u_{i+1}}$$

- Par récurrence, on établit ensuite :

$$c_{u_0} \leqslant 2p + c_{u_p}$$

Exponentiation rapide

En procédant de la même façon avec l'inégalité $1 + c_{\lfloor n/2 \rfloor} \leqslant c_n$, on aboutit à l'encadrement suivant.

$$\forall n \in \mathbb{N} \qquad 2 + \lfloor \log_2 n \rfloor \leqslant c_n \leqslant 2 + 2\lfloor \log_2 n \rfloor$$

La *complexité temporelle de l'exponentiation rapide* est :

$$c_n = \Theta(\log_2 n)$$

Pour $n \in \mathbb{N}^*$, en notant n_n le nombre de bits égaux à 1 dans la décomposition binaire de n, on a :

$$c_n = \lfloor \log_2 n \rfloor + n_n$$

Remarque

La notation Θ indique que c_n est minorée et majorée par deux fonctions affines de $\log_2 n$. Elle permet une connaissance plus fine du comportement asymptotique de la complexité du code.

Paradigme DPR

Paradigme DPR

Les exemples précédents illustrent un paradigme de programmation appelé *diviser pour régner* (DPR).

Son principe consiste à diviser un problème caractérisé par un entier n en sous-problèmes identiques caractérisés par des entiers αn avec $\alpha < 1$. Souvent, $\alpha = 1/2$.

L'un des principaux avantages de DPR est la *réduction du coût temporel* de traitement par rapport à d'autres algorithmes.

Sa mise en œuvre fait naturellement appel à la *récursivité*.

Relation de récurrence

Considérons un algorithme mettant en œuvre le paradigme DPR en partageant un problème de taille n en deux sous-problèmes de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$.

Le coût temporel c_n de l'algorithme vérifie une relation de récurrence de la forme :

$$c_n = ac_{\lfloor n/2 \rfloor} + bc_{\lceil n/2 \rceil} + d_n$$

avec $a + b \geqslant 1$. d_n représente le coût du partage et de la recomposition du problème.

Récurrence avec $n = 2^p$

Si $n = 2^p$, on pose $u_p = c_{2^p}$. La relation de récurrence devient alors :

$$\frac{u_p}{(a+b)^p} = \frac{u_{p-1}}{(a+b)^{p-1}} + \frac{d_{2^p}}{(a+b)^p}$$

Par télescopage, on obtient :

$$u_p = (a+b)^p \left(u_0 + \sum_{j=1}^{p} \frac{d_{2^j}}{(a+b)^j} \right)$$

Récurrence avec $n = 2^p$

Pour aller plus loin dans le calcul, d_n doit être connu. Dans la suite, nous supposons $d_n = \lambda n^k$ (coût polynomial au sens large).

Alors :

$$u_p = \begin{cases} \alpha 2^{kp} + \beta(a+b)^p & \text{si } a+b \neq 2^k \\ (u_0 + \lambda p)(a+b)^p & \text{si } a+b = 2^k \end{cases}$$

avec $\alpha = \lambda 2^k / (2^k - a - b)$ et $\beta = u_0 - \alpha$.

Récurrence avec $n = 2^p$

Finalement :

- Si $a + b < 2^k$ alors $u_p \sim \alpha 2^{kp}$. D'où : $c_n \sim \alpha n^k$.
- Si $a + b = 2^k$ alors $u_p \sim \lambda p 2^{kp}$. D'où : $c_n \sim \lambda n^k \log n$.
- Si $a + b > 2^k$ alors $u_p \sim \beta(a + b)^p$. D'où : $c_n \sim \beta n^{\log(a+b)}$.

Cas général

Pour établir un résultat dans le cas général, on montre tout d'abord que *si la suite* $(d_n)_{n\in\mathbb{N}}$ *est croissante, alors la suite* $(c_n)_{n\in\mathbb{N}}$ *l'est aussi*.

On sait que tout entier naturel n peut être *encadré par deux puissances consécutives de 2* :

$$2^p \leqslant n < 2^{p+1}$$

avec $p = \lfloor \log n \rfloor$.

Alors : $u_p \leqslant c_n \leqslant u_{p+1}$.

Cas général

Si $d_n = \lambda n^k$, le résultat suivant est appelé *théorème maître*.

- Si $\log(a+b) < k$ alors $c_n \in \Theta(n^k)$.
- Si $\log(a+b) = k$ alors $c_n \in \Theta(n^k \log n)$.
- Si $\log(a+b) > k$ alors $c_n \in \Theta(n^{\log(a+b)})$.

Autres exemples

De nombreuses situations font appel à une résolution par la technique *diviser pour régner*.

- Rercherche d'un élement dans un tableau bi-dimensionnel ordonné
- Produits de matrices
- Plus proches voisins dans un nuage de points du plan
- Tri-fusion