LOG1810
8 - Techniques de dénombrement avancées

Lévis Thériault, Aurel Randolph et Eric Demers
Merci à Chris Kauffman

Dernière mise à jour:
2025-03-16

Logistique

Lecture: Rosen

- Maintenant: 8.1 - 8.6
- Suivant: 10.1 - 10.8

Objectifs

- Relations de récurrence
- Fonctions génératrices

Notions de base

- Une **relation de récurrence** définit une relation entre les éléments d'une séquence
- Ex: La séquence de Fibonacci *satisfait* la relation de récurrence
$$f_n = f_{n-1} + f_{n-2}$$
- Remarque : selon certains témoignages, cette relation est également la **définition** de Fibonacci mais à l'origine, la relation a été découverte pour étudier la croissance d'une **population de lapins** idéalisée
- Les relations de récurrence peuvent être exprimées dans une variété de notations telles que
$$f(n) = f(n-1) + f(n-2)$$
$$g(n) = 3g(n-1) + 4g(n/5)$$

Temps d'exécution d'algorithme et relations de récurrence

- Nous examinerons plusieurs algorithmes et montrerons que leurs temps d'exécution peuvent être représentés comme des relations de récurrence
- Nous examinerons comment caractériser les relations de récurrence afin que les temps d'exécution des algorithmes puissent être estimés

Recherche binaire

- Entrée: tableau de taille `n`
- À l'itération 0 faire 8-10 op. pour réduire la taille à `n/2`
- À l'itération 1 faire 8-10 op. pour réduire la taille à `n/4`
- Dans le pire des cas, la `clé` n'est pas dans le tableau donc réduction à 0 élément
- Total op. dans le pire des cas est décrit par la relation de récurrence

$$f(n) = f(n/2) + c$$

où c est une constante

```
int binary_search(int a[], int key){
  int left=0, right=a.length-1;
  int mid = 0;
  while(left <= right){
    mid = (left+right)/2;
    if(key == a[mid]){
      return mid;
    }else if(key < a[mid]){
      right = mid-1;
    }
    else{
      left = mid+1;
    }
  }
  return -1;
}
```

Exercice: Recherche du point milieu

- Recherche récursive d'un élément dans un tableau non trié
- En quoi est-ce différent de la recherche binaire ?
- Développer une relation de récurrence pour le nombre d'opérations utilisées pour un tableau de taille n
- À quoi vous attendez-vous pour l'exécution de cet algorithme ?

```
// Determine if key is present in
// UNSORTED array a[] by repeated
// bisection search
boolean midpoint_search(int a[], int key)
  int left=0, right=length(a)-1;
  return helper(a,key,left,right);
}
boolean helper(int a[], int key,
               int left, int right)
{
  if(left > right){
    return false;
  }
  int mid = (left+right)/2;
  if(key == a[mid]){
    return true;
  }
  boolean foundL,foundR;
  foundL = helper(a,key,left,mid-1);
  foundR = helper(a,key,mid+1,right);
  reutrn foundL OR foundR;
}
```

Réponses: Recherche du point milieu

- En quoi est-ce différent de la recherche binaire?
  - Point milieu va à gauche **et** à droite
  - Binaire va à gauche **ou** à droite
- Développons une relation de récurrence pour nb. d'opérations.
  - Moitié du tableau mais va à la fois gauche/droite
  - Utilise O(c) d'opérations pour réduire de moitié
  - $f(n) = 2f(n/2) + c$
- À quoi vous attendez-vous pour l'exécution de cet algorithme ?
  - Visite chaque élément du tableau une fois donc le pire des cas linéaire $O(n)$

```
// Determine if key is present in
// UNSORTED array a[] by repeated
// bisection search
boolean midpoint_search(int a[], int key){
  int left=0, right=length(a)-1;
  return helper(a,key,left,right);
}
boolean helper(int a[], int key,
               int left, int right)
{
  if(left > right){
    return false;
  }
  int mid = (left+right)/2;
  if(key == a[mid]){
    return true;
  }
  boolean foundL,foundR;
  foundL = helper(a,key,left,mid-1);
  foundR = helper(a,key,mid+1,right);
  reutrn foundL OR foundR;
}
```

Tri fusion (Merge Sort)

- Comprend deux phases
  - **Division descendante** d'un tableau en deux moitiés, s'arrête en atteignant les tableaux de taille 1
  - **Fusion ascendante** de deux tableaux triés dans un tableau plus grand
- Examinons les deux brièvement pour fins d'analyse

Exercice: Opération de fusion

- Fusionne deux tableaux triés en un tableau trié combiné
- Montre comment ça marche sur `a[]={1,3,5,9}; b[]={2,3,6}`
- Quelle est la complexité d'exécution de `merge()` ?

```
// Merge sorted arrays a[] and b[] int res[]
// which is also sorted
void merge(int[] res, int[] a, int[] b){
  int ai=0, bi=0;
  for(int ri=0; ri<length(res); ri++){
    if(ai >= length(a)){     // a[] gone
      res[ri] = b[bi];
      bi++;
    }
    else if(bi >= length(b)){// b[] gone
      res[ri] = a[ai];
      ai++;
    }
    else if(a[ai]<=b[bi]){   // a[] smaller
      res[ri] = a[ai];
      ai++;
    }
    else{                   // b[] smaller
      res[ri] = b[bi];
      bi++;
    }
  }
}
```

Réponses: Opération de fusion

- Fusionne deux tableaux triés en un tableau trié combiné
- Montre comment ça marche sur `a[]={1,3,5,9}; b[]={2,3,6}`
- Quelle est la complexité d'exécution de `merge()` ?
  - Temps linéaire $O(n)$ en taille du tableau `res[]` qui est la somme des longueurs de `a[]` et `b[]`

```
// Merge sorted arrays a[] and b[] int res[]
// which is also sorted
void merge(int[] res, int[] a, int[] b){
  int ai=0, bi=0;
  for(int ri=0; ri<length(res); ri++){
    if(ai >= length(a)){     // a[] gone
      res[ri] = b[bi];
      bi++;
    }
    else if(bi >= length(b)){// b[] gone
      res[ri] = a[ai];
      ai++;
    }
    else if(a[ai]<=b[bi]){   // a[] smaller
      res[ri] = a[ai];
      ai++;
    }
    else{                   // b[] smaller
      res[ri] = b[bi];
      bi++;
    }
  }
}
```

Exercice: Tri fusion, diviser, fusionner

- Le tri fusion utilise une descente récursive qui divise par 2 le tableau
- A l'atteinte d'un tableau de taille 0 ou 1 arrêts de récursivité : ces tableaux sont "triés"
- Fusionner les tableaux lors de la remonté de la récursion.

```
void merge_sort(int[] a) {
  if (length(a) <= 1) {
    return;
  }
  int len = length(a);
  int[] left  = array_copy(a, 0,     len/2);
  int[] right = array_copy(a, len/2, len);

  merge_sort(left);
  merge_sort(right);

  merge(a, left, right);
}
```

Questions

- Quelle est la complexité de `array_copy()` ?
- Quelle est la complexité de `merge()` ?
- Donnez une relation de récurrence pour le total des opérations effectuées par `merge_sort()`

Réponses: Tri fusion, diviser, fusionner

- Quelle est la complexité de `array_copy()`?
  - Linéaire $O(n)$
- Quelle est la complexité de `merge()`?
  - Du dernier exercice était $O(n)$
- Donnez une relation de récurrence pour tous les op. de `merge_sort()`
  - Récursion sur la moitié: $f(n/2)$
  - Récursion des deux côtés: $2f(n/2)$
  - Travail linéaire à chaque étape pour copier/fusionner

$$f(n) = 2f(n/2) + a \cdot n + b$$

```
void merge_sort(int[] a) {
  if (length(a) <= 1) {
    return;
  }
  int len = length(a);
  int[] left  = array_copy(a, 0, len/2);
  int[] right = array_copy(a, len/2, len);

  merge_sort(left);
  merge_sort(right);

  merge(a, left, right);
}
```

Théorème maître

Soit f une fonction croissante qui vérifie la relation de récurrence

$$f(n) = af(n/b) + cn^d$$

- quelque soit $n = b^k$, avec k comme entier positif
- $a \geq 1$
- $b \geq 1$ et est un entier
- $c > 0$ et $d \geq 0$ nombres réels

Alors $f(n)$ tombe dans l'une des classes de complexité suivantes

$$\begin{array}{lll} \text{(Cas 1)} & O(n^d) & \text{pour } a < b^d \\ \text{(Cas 2)} & O(n^d \log n) & \text{pour } a = b^d \\ \text{(Cas 3)} & O(n^{\log_b a}) & \text{pour } a > b^d \end{array}$$

- La preuve est donnée en exercices dans le texte et nous ne nous y attarderons pas
- La question pratique est qu'il permet une analyse BEAUCOUP plus facile de la récurrence / algorithmes diviser pour régner

Exercice: Analyse des algorithmes

Applications du théorème maître :

$$f(n) = af(n/b) + cn^d$$

1. $f(n) = O(n^d)$ si $a < b^d$
2. $f(n) = O(n^d \log n)$ si $a = b^d$
3. $f(n) = O(n^{\log_b a})$ si $a > b^d$

Recherche binaire

Le total des opérations dans le pire des cas est décrit en récurrence $f(n) = f(n/2) + q$ avec q une constante

- $a = 1, b = 2, d = 0$
- Par le théorème maître, Cas 2
- $O(n^0 \log n) = O(\log n)$

Recherche du point milieu

- $f(n) = 2f(n/2) + q$
- Analyser et déterminer le nombre d'opérations Grand-O

Tri fusion

- $f(n) = 2f(n/2) + q \cdot n + w$
- Analyser et déterminer le nombre d'opérations Grand-O

Réponses: Analyse des algorithmes

Théorème maître

$$f(n) = af(n/b) + cn^d$$

1. $f(n) = O(n^d)$ si $a < b^d$
2. $f(n) = O(n^d \log n)$ si $a = b^d$
3. $f(n) = O(n^{\log_b a})$ si $a > b^d$

Recherche binaire

Nb. total d'op. dans le pire des cas est décrit en récurrence $f(n) = f(n/2) + q$ avec q une constante

- $a = 1, b = 2, d = 0$
- Théorème maître, Cas 2
- $O(n^0 \log n) = O(\log n)$

Recherche du point milieu

$f(n) = 2f(n/2) + q$

- $a = 2, b = 2, d = 0$
- Théorème maître, Cas 3
- $O(n^{\log_2(2)}) = O(n)$

Tri fusion

$f(n) = 2f(n/2) + q \cdot n + w$

- $a = 2, b = 2, d = 1$
- Théorème maître, Cas 2
- $O(n^d \log n) = O(n^1 \log n)$

Exercice: D'autres types de relations de récurrence

Théorème maître

$$f(n) = af(n/b) + cn^d$$

$$\begin{array}{lll} \text{(Cas 1)} & O(n^d) & \text{pour } a < b^d \\ \text{(Cas 2)} & O(n^d \log n) & \text{pour } a = b^d \\ \text{(Cas 3)} & O(n^{\log_b a}) & \text{pour } a > b^d \end{array}$$

À laquelle des relations de récurrence suivantes le théorème maître s'applique-t-il et à laquelle ne s'applique-t-il pas ?

1. $f(n) = f(n-1) + 7$
2. $f(n) = 3 \cdot f(n-1)$
3. $f(n) = 4 \cdot f(n/2) + 10$
4. $f(n) = f(n-1) + f(n-2)$

Pourquoi le théorème maître s'applique-t-il à certains et pas à d'autres ?

Réponses: D'autres types de relations de récurrence

Théorème maître

$$f(n) = af(n/b) + cn^d$$

$$\begin{array}{lll} (\text{Cas } 1) & O(n^d) & \text{for } a < b^d \\ (\text{Cas } 2) & O(n^d \log n) & \text{for } a = b^d \\ (\text{Cas } 3) & O(n^{\log_b a}) & \text{for } a > b^d \end{array}$$

À laquelle des relations de récurrence suivantes le théorème maître s'applique-t-il et à laquelle ne s'applique-t-il pas ?

1.	$f(n) = f(n-1) + 7$	Non, linéaire RR, degré 1
2.	$f(n) = 3 \cdot f(n-1)$	Non, linéaire RR, degré 1
3.	$f(n) = 4 \cdot f(n/2) + 10$	Oui, diviser/régner RR, $O(n)$
4.	$f(n) = f(n-1) + f(n-2)$	Non, linéaire RR, degré 2

- Le théorème maître s'applique aux algorithmes **diviser pour régner** et leurs relations de récurrence associées
- Nécessite une récurrence impliquant une **division**
- 1, 2 et 4 sont des **relations de récurrence linéaire**

Exercice: Proposez une solution

Pour les relations de récurrence suivantes

- Calculez $f(5)$
- Donnez une solution sous forme explicite pour la relation de récurrence
  - Une solution qui n'implique pas de récurrence

Relation de récurrence 1

Récurrence $f(n) = f(n-1) + 7$
Cas de base $f(0) = 0$

- $f(5) = f(4) + 7 = \ldots$

Relation de récurrence 2

Récurrence $f(n) = 3 \cdot f(n-1)$
Cas de base $f(0) = 1$

- $f(5) = 3 \cdot f(4) = \ldots$

Réponses: Proposez une solution

Pour les relations de récurrence suivantes

- Calculez $f(5)$
- Donnez une solution explicite pour la relation de récurrence

Relation de récurrence 1

Récurrence $f(n) = f(n-1) + 7$

Cas de base $f(0) = 0$

$$\begin{aligned} f(5) &= 7 + f(4) \\ &= 7 + 7 + f(3) \\ &= 7 + 7 + 7 + f(2) \\ &= 7 + 7 + 7 + 7 + f(1) \\ &= 7 + 7 + 7 + 7 + 7 + f(0) \\ &= 7 + 7 + 7 + 7 + 7 + 0 \\ &= 7 \cdot 5 \\ f(n) &= 7 \cdot n \end{aligned}$$

Relation de récurrence 2

Récurrence $f(n) = 3 \cdot f(n-1)$

Cas de base $f(0) = 1$

$$\begin{aligned} f(5) &= 3 \cdot f(4) \\ &= 3 \cdot 3 \cdot f(3) \\ &= 3 \cdot 3 \cdot 3 \cdot f(2) \\ &= 3 \cdot 3 \cdot 3 \cdot 3 \cdot f(1) \\ &= 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot f(0) \\ &= 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 1 \\ &= 3^5 \\ f(n) &= 3^n \end{aligned}$$

Les relations de récurrence linéaires homogènes ont des solutions générales

- Les relations de récurrence linéaires **homogènes** ont la forme suivante avec des constantes r_i

$$
\begin{aligned}
f(n) &= a_1 f(n-1) + a_2 f(n-2) + \ldots + a_k f(n-k) \\
&= \alpha_1 r_1^n + \alpha_2 r_2^n + \ldots + \alpha_k r_k^n
\end{aligned}
$$

- La solution consiste généralement à déterminer $r_1, r_2, \ldots, r_k$ comme racines de l'**équation caractéristique** associée

$$r^k - a_1 r^{k-1} - a_2 r^{k-2} - \ldots - a_k r^{k-k} = 0$$

- Les $r_1, r_2, \ldots$ connues, déterminez les coefficients $\alpha_1, \alpha_2, \ldots$ en résolvant les conditions initiales

Exemple de résolution d'une RR linéaire homogène

Résoudre la RR linéaire homogène suivante

ÉTANT DONNÉ	
Récurrence:	$f(n) = f(n-1) + 2f(n-2)$
	Alors degré 2 avec $a_1 = 1, a_2 = 2$
Cas de base:	$f(0) = 2,\ f(1) = 7$
Sol. forme gén.:	$f(n) = \alpha_1 r_1^n + \alpha_2 r_2^n$
Résoudre les r_i	
Eq. car.:	$r^2 - a_1 r - a_2 = 0$ avec $a_1 = 1, a_2 = 2$
Racine de l'éq. car.:	$r^2 - r - 2 = 0$
	$(r-2)(r+1) = 0$ donc racines 2,-1
Sub. dans forme gén.:	$f(n) = \alpha_1 (2)^n + \alpha_2 (-1)^n$
Résoudre les α_i	
Utilisez cond. init.	$f(0) = \alpha_1 \cdot 2^0 + \alpha_2 \cdot (-1)^0 = \alpha_1 + \alpha_2 = 2$
résoudre α_i:	$f(1) = \alpha_1 \cdot 2^1 + \alpha_2 \cdot (-1)^1 = 2\alpha_1 - \alpha_2 = 7$
	2 équations linéaires à 2 inconnues, α_1, α_2
	Résoudre pour obtenir $\alpha_1 = 3, \alpha_2 = -1$
Solution finale:	$f(n) = 3 \cdot 2^n - (-1)^n$

Au-delà des relations de récurrence linéaires homogènes

- Généralement les relations de récurrence qui sont basées sur les k derniers termes sont exponentielles
- En utilisant ce cadre, il est possible de démontrer que le nième nombre de Fibonacci est

$$fib(n) = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n$$

  qui est exponentiel
- Il est possible de résoudre des relations de récurrence **non homogènes** plus complexes qui incluent des fonctions $g(n)$ non récurrentes par l'utilisation des fonctions génératrices.

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k) + \mathbf{g(n)}$$

- La résolution est plus délicate et nécessite un élargissement des techniques utilisées pour les récurrences linéaires

En résumé

- On peut résoudre exactement certains types de relations de récurrence, en particulier les relations de récurrence linéaire
- Les relations de récurrence sont importantes afin d'estimer le nombre d'opérations des algorithmes diviser pour conquérir
- En particulier, le théorème maître permet d'obtenir un estimé Grand-O pour beaucoup de ceux-ci

Annexe

Éléments de preuve pour le théorème maître

Éléments de preuve pour le théorème maître

Soit la relation de récurrence

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ aT(n/b) + g(n) & \text{si } n > 1 \end{cases}$$

- Le problème est subdivisé en sous-problèmes de tailles n/b plusieurs fois.
- Utilisons la substitution répétée afin de démontrer le théorème maître lorsque $g(n) = n^d$

Éléments de preuve pour le théorème maître, substitution répétée

- La relation de départ est $T(n) = aT(n/b) + n^d$
- Substituer $T(n/b) = aT(n/b^2) + (n/b)^d$ une première fois permet d'obtenir:

$$T(n) = a(aT(n/b^2)+(n/b)^d)+n^d = a^2T(n/b^2)+n^d(1+a/b^d)$$

- Substituer ensuite $T(n/b^2) = aT(n/b^3) + (n/b^2)^d$ permet d'obtenir:

$$T(n) = a^3T(n/b^3) + n^d(1 + a/b^d + (a/b^d)^2)$$

- Nous pouvons alors supposer que:

$$T(n) = a^jT(n/b^j) + n^d\sum_{i=0}^{j-1}(a/b^k)^i$$

Éléments de preuve pour le théorème maître, substitution répétée

- Ensuite en subdivisant jusqu'à obtenir $n = 1$:

$$T(n) = a^j c + n^d \sum_{i=0}^{j-1} (a/b^k)^i$$

- Sachant qu'il faut subdiviser $\log_b n$ fois pour arriver à $n = 1$ nous pouvons écrire:

$$T(n) = ca^{\log_b n} + n^d \sum_{i=0}^{j-1} (a/b^k)^i$$

- ou encore puisque $a^{\log_b n} = n^{\log_b a}$

$$T(n) = cn^{\log_b a} + n^d \sum_{i=0}^{j-1} (a/b^k)^i$$

Éléments de preuve pour le théorème maître, cas 2

(Cas 2) Si $a = b^d$

- $T(n) = cn^{\log_b(b^d)} + n^d \sum_{i=0}^{j-1}(1)^i$
- $T(n) = cn^d + n^d j$
- $T(n) = cn^d + n^d \log_b n$

Puisque le deuxième terme est d'ordre plus grand

- $T(n) = O(n^d \log n)$

Éléments de preuve pour le théorème maître, cas 1 et 3

- Maintenant utilisant l'identité suivante :

$$\sum_{k=0}^{n-1} ar^k = a\frac{1-r^n}{1-r}$$

- Le dernier terme devient:

$$T(n) = cn^{\log_b a} + n^d \frac{1-(a/b^d)^{\log_b n}}{1-a/b^d}$$

- Puisque $(a/b^d)^{\log_b n} = a^{\log_b n}/(b^{\log_b n})^d = n^{\log_b a}/n^d$ alors:

$$T(n) = cn^{\log_b a} + n^d \frac{1-n^{\log_b a}/n^d}{1-a/b^d}$$

Éléments de preuve pour le théorème maître, cas 1 et 3

- Que nous pouvons réécrire:

$$T(n) = n^{\log_b a}\left(c - \frac{1}{1 - a/b^d}\right) + n^d \frac{1}{1 - a/b^d}$$

- (Cas 1) Si $\log_b a < d$ alors le deuxième terme est d'ordre plus grand $T(n) = O(n^d)$
- (Cas 3) Si $\log_b a > d$ alors le premier terme est d'ordre plus grand $T(n) = O(n^{\log_b a})$