Cours 1: Introduction à l'algorithmique

Olivier Bournez

bournez@lix.polytechnique.fr
LIX, Ecole Polytechnique

Aujourd'hui

Calcul de x^n

Maximum

Complexité d'un problème

Problème du maximum

Trier

Recherche d'un algorithme pour calculer x^n.

- On part avec
  - $y_0 = x$ (un entier, un réel, une matrice, ...)
  - et un entier n.
- On souhaite calculer x^n le plus rapidement possible.

- Plus formellement:
  - On doit construire une suite $y_1, y_2, \cdots, y_k$.
  - Règle du jeu: si j'ai déjà calculé $y_0, y_1, \cdots, y_{i-1}$, alors je peux calculer $y_i = y_j * y_k$, pour $0 \leq j, k < i$.
  - Question: trouver le meilleur algorithmque:

$$opt(n) = min\{r | \exists y_0, y_1, \cdots, y_r \text{ suite correcte avec } y_r = x^n\}.$$

Première solution: Algorithme naïf

- On peut le faire avec $n - 1$ étapes.
- En effet, il suffit de faire:
  - $y_0 = x$,
  - pour $i = 1$ à n,
    - $y_i = y_{i-1} * y_0$.

Méthode plus astucieuse: méthode binaire

- Plus astucieux:
  - n pair, $x^n = (x^{n/2})^2$.
  - n impair, $x^n = (x^{\lfloor n/2 \rfloor})^2 * x$.
  - $n = 0$, $x^n = 1$
  - $n = 1$, $x^n = x$.
- D'où le programme:

```
fonction carre(int x) {
retourner x*x
}

fonction puissance (int x, int n) {
si n==0 alors retourner 1
si n==1 alors retourner x
si n est pair alors
      retourner carre(puissance(x,n div 2))
si n est impair alors
      retourner carre(puissance(x, n div 2))*x
}
```

- Est-ce le meilleur algorithme?
- Réponse non:
  - (plus petit contre exemple:) 15.
  - Pour 15, méthode binaire prend 6 multiplications.
  - On peut faire mieux: $y = x^3$ se calcule en deux multiplications, et $y^5 = x^{15}$ en trois multiplications, soit un total de 5.

Reformulation de la méthode binaire

- Sur un exemple:
  - 13 en binaire: 1101.
    - on barre le premier 1
    - pour chaque 1, on fait $carre(x) * x$: note cx
    - pour chaque 0, on fait $carre(x)$: note c.
  - Ici: 1101 donne $cxccx$:
    - on calcule x^2,x^3, x^6, x^{12}, x^{13}.
- Cout $cc(n)$ pour calculer x^n par cette méthode: nombre de chiffres en base 2 de n + nombre de 1 - 2.
- nb de chiffres en base 2 de n: $\lfloor \log_2 n \rfloor + 1$
- On obtient:

$$\lfloor \log_2 n \rfloor \leq cc(n) \leq 2\lfloor \log_2 n \rfloor.$$

Retour sur le contre-exemple

- $15 = 1111$ en binaire
- $cc(15) = 6$.
- 15 = 5*3.
  - $y^1 = x^2$, $y^2 = x^3$.
  - $y^3 = y^1 * y^1 = x^4$, $y^4 = y^3 * x = x^5$.
  - $y^5 = y^2 * y^4$.
- Idée: méthode des facteurs.

Méthode des facteurs

- On considère $n = p * q$, où p est le plus petit facteur premier de n.
- On calcule x^n en calculant x^p et en élevant le résultat à la puissance q.
- Si n est premier, on calcule x^{n-1} et on multiplie par x.
- Sur un exemple:
  - $55 = 5 * 11$.
  - On calcule $y = x^5 = x^4 * x = (x^2)^2 * x$
  - On calcule $y^{11} = y^{10} * y = (y^2)^5 * y$.
  - Au total 8 multiplications.

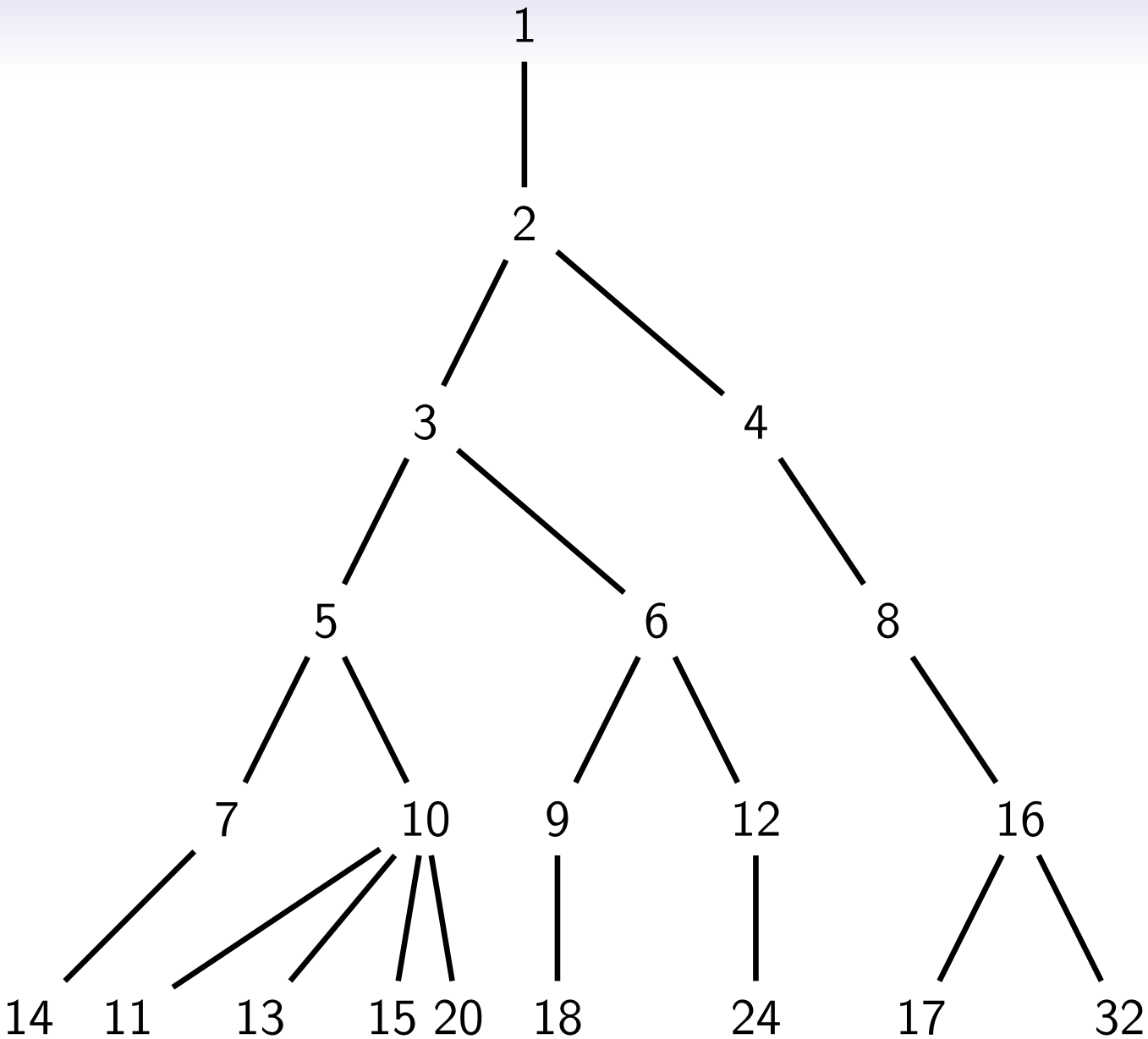
- Est-ce le meilleur algorithme?
- Réponse non:
  - (plus petit contre exemple:) 33.
    - la méthode le fait en 7 multiplications,
    - mais on peut le faire en 6 (par la méthode binaire).

- Il y a une infinité de nombres pour lesquels la méthode des facteurs est meilleure que la méthode binaire (prendre $n = 15 * 2^k$) ou le contraire (prendre $n = 33 * 2^k$).

- Pour les puissances de 2, les deux méthodes sont identiques.

Méthode de l'arbre de Knuth

- On construit un arbre.
  - On part de (la racine) 1 (donc au niveau 1).
  - Supposons construit les k premiers niveaux. on prend chaque sommet n du $k$ème niveau, de gauche à droite, et on le relie avec les sommets $n+1$, $n+a_1$, $n+a_2$, $\ldots$, $n+a_{k-1}=2n$ (dans cet ordre), où $1, a_1, a_2, \ldots, a_{k-1}=n$ représente le chemin de la racine à n.
  - On ajoute pas un sommet si il est déjà dans l'arbre.
- Arbre obtenu sur le transparent qui suit pour les premiers niveaux.
- Intérêt:
  - Le chemin menant de la racine à n indique une suite d'exposants permettant de calculer x^n de façon efficace.


1
2
3
4
5
6
8
7
10
9
12
16
14
11
13
15
20
18
24
17
32

Statistiques

- Optimal? Non, mais ...
- (Plus petit contre exemples:) 77 (puis 154, et 233).
- Plus petit n tel que méthode de l'arbre meilleur que méthode des facteurs et méthode binaire est $n = 23$.
- Plus petit n pour lequel la la méthode de l'arbre est moins bonne que celle des facteurs est $n = 19879 = 103 * 194$.
- Pour $n \leq 100000$, l'arbre est moins bon que les deux autres méthodes pour seulement 6 entiers.

Borne inférieure

- Théorème. $\lfloor \log_2 n \rfloor \leq opt(n) \leq 2\lfloor \log_2 n \rfloor$.
- Par récurrence: la suite $y_0 = x, \ldots, y_r$ est telle que $y_k = x^{\alpha(k)}$.
- Par récurrence $\alpha(k) \leq 2^k$ car:
  - c'est vrai pour $k = 0$.
  - $y_k = y_i * y_j => \alpha(k) = \alpha(i) + \alpha(j)$.
  - HR: $\alpha(i) \leq 2^i$ et $\alpha(j) \leq 2^j$.
  - Donc $\alpha(k) \leq 2^i + 2^j \leq 2 * 2^{\max\{i,j\}}$.
  - et puisque $i, j \leq k - 1$, $\alpha(k) \leq 2^k$.
- Pour le nombre d'étapes k de l'algorithme optimal, on doit avoir $\alpha(k) = n$, et donc $\lfloor \log_2 n \rfloor \leq k = opt(n)$.
- Calculer $opt(n)$ est difficile (NP-complet).
- On peut prouver que $\lim_{n\to\infty} \frac{opt(n)}{\log_2 n} = 1$ (voir Knuth + cours de Yves Robert).

Aujourd'hui

Calcul de x^n

Maximum

Complexité d'un problème

Problème du maximum

Trier

Calculer le maximum

- Le problème *MAX*.
  - On se donne une liste d'entiers naturels $e_1, e_2, \cdots, e_n$.
  - On veut calculer $\max\{e_1, e_2, \cdots, e_n\}$.

Calculer le maximum

- Le problème *MAX*.
  - On se donne une liste d'entiers naturels $e_1, e_2, \cdots, e_n$.
  - On veut calculer $\max\{e_1, e_2, \cdots, e_n\}$.

-1 si la liste est vide.

```
int [] T= new int[100];
int r = max(T);

...

int max(int [] T) {
 int max = T[1];
 for (int j=2; j<= T.length; j++) {
      if (T[j] > max)
        max = T[j];}
 return max;
}
```

- Nombre de comparaisons (hors variable de boucle)
  $C(n) = n - 1$.
- Nombre d'affectations (hors variable de boucle) $1 \leq B(n) \leq n$.
  $n =$ longueur de la liste.

Bornes extrêmes

- $B(n) = 1$ pour une liste qui débute par son plus grand élément.
- $B(n) = n$ pour une liste triée.

- $B(n)$ en moyenne?

  - Attention, pour parler de moyenne, il faut que cela ait un sens.
  - Par exemple, en supposant que les entrées sont des permutations de $1, 2, \ldots, n$, et que ces permutations sont équiprobables.
  - $1 \leq B(n) \leq n$.

Aujourd'hui

Calcul de x^n

Maximum

Complexité d'un problème

Problème du maximum

Trier

Complexité d'un problème, d'un algorithme

- On fixe une mesure élémentaire:
  - nombre de comparaisons,
  - nombre d'affectations,
  - mémoire utilisée,
  - temps utilisé,
  - ...
- Cette mesure associe à un algorithme $\mathcal{A}$ et une entrée d une valeur
$$\mathit{Complexite}(\mathcal{A}, d).$$
- On cherche souvent à évaluer cette complexité en fonction d'un paramètre naturel $n = \mathit{taille}(d)$, représentatif des entrées.
- Exemple:
  - $\mathcal{A}$: l'algorithme itératif pour *MAX* précédent.
  - d: une liste donnée en entrée.
  - $n = \mathit{taille}(d)$, le nombre d'éléments dans la liste.
  - $\mathit{Complexite}(\mathcal{A}, d)$, le nombre de comparaisons de $\mathcal{A}$ sur d.

- On peut alors parler de la complexité (au pire cas)
  - d'un algorithme (on fait varier seulement les entrées)

$$Complexite_{\mathcal{A}}(n) = \max_{d\,/\,taille(d)=n} Complexite(\mathcal{A}, d).$$

  - d'un problème (on fait varier l'algorithme, et les entrées) on considère le meilleur algorithme qui résoud le problème. Ce n'est pas (subtile):

$$Complexite(n) = \inf_{\mathcal{A}\ \text{correct}} \max_{d\,/\,taille(d)=n} Complexite(\mathcal{A}, d)$$

- On arrive parfois à parler de la complexité exacte.
- On doit souvent se limiter à une étude asymptotique, ou à des bornes asymptotiques.

Aujourd'hui

Calcul de x^n

Maximum

Complexité d'un problème

Problème du maximum

Trier

Exemple d'étude exacte

- Etude du problème *MAX* en nombre de comparaisons:
  - Le problème *MAX* admet une solution avec $n - 1$ comparaisons.

Exemple d'étude exacte

- Etude du problème *MAX* en nombre de comparaisons:
  - Le problème *MAX* admet une solution avec $n-1$ comparaisons.
  - Cet algorithme est optimal en nombre de comparaisons.

Preuve

- Proposition: dans tout algorithme, tout élément autre que le maximum doit être comparé au moins une fois avec un élément qui lui est plus grand.

- Conséquence: il faut au moins $n - 1$ comparaisons pour résoudre MAX.

Preuve

- Proposition: dans tout algorithme, tout élément autre que le maximum doit être comparé au moins une fois avec un élément qui lui est plus grand.
- Preuve:
  - soit i_0 le rang du maximum M retourné par l'algorithme sur une liste $L = e_1.e_2.\cdots.e_n$.
  - Par l'absurde: soit $j_0 \neq i_0$ tel que e_{j_0} n'est pas comparé avec un élément plus grand que lui.
  - e_{j_0} n'a donc pas été comparé avec e_{i_0} le maximum
  - Considérer la liste $L' = e_1.e_2.\cdots.e_{j_0-1}.M+1.e_{j_0+1}.\cdots.e_n$ obtenue à partir de L en remplaçant l'élément d'indice j_0 par $M+1$.
  - L'algorithme effectue exactement les mêmes comparaisons sur L et L', sans comparer $L'[j_0]$ avec $L'[i_0]$ et donc retournera $L'[i_0]$, ce qui est incorrect.
- Conséquence: il faut au moins $n-1$ comparaisons pour résoudre MAX.

Asymptotiques

- Pour un processeur capable d'effectuer un million d'instructions élémentaires par seconde.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	4 s
$n = 30$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	18 m	10^{25} a
$n = 50$	< 1 s	< 1 s	< 1 s	< 1 s	11 m	36 a	∞
$n = 10^2$	< 1 s	< 1 s	< 1 s	1s	12.9 a	10^{17} a	∞
$n = 10^3$	< 1 s	< 1 s	1s	18 m	∞	∞	∞
$n = 10^4$	< 1 s	< 1 s	2 m	12 h	∞	∞	∞
$n = 10^5$	< 1 s	2 s	3 h	32 a	∞	∞	∞
$n = 10^6$	1s	20s	12 j	31710 a	∞	∞	∞

- Notations: ∞ = le temps dépasse 10^{25} années, s= seconde, m= minute, h = heure, a = an.

Notation (Pire Cas)

- Notation:
  - $f(n) = O(g(n))$ si et seulement si il existe deux constantes positives n_0 et B telles que

    $$\forall n \geq n_0, f(n) \leq Bg(n)$$

    Ce qui signifie que f ne croît pas plus vite que g.

- Exemple:
  - Le problème *MAX* admet une solution itérative en $O(n)$ affectations.
  - L'algorithme itératif précédent fonctionne en $O(n)$ affectations.

- Un algorithme
  - en temps $O(1)$ effectue un nombre constant d'opérations.
  - en temps $O(n)$ s'appelle un algorithme linéaire.
  - en temps $O(n^k)$ s'appelle un algorithme polynomial.

Notation

- Notation (suite)
  - $f(n) = \Omega(g(n))$ si et seulement si il existe deux constantes positives n_0 et B telles que

$$\forall n \geq n_0, f(n) \geq Bg(n)$$

  - $f(n) = \Theta(g(n))$ si et seulement si il existe trois constantes positives n_0, B et C telles que

$$\forall n \geq n_0, Bg(n) \leq f(n) \leq Cg(n)$$

- Exemple:
  - Le problème MAX nécessite $\Theta(n)$ comparaisons.
  - (à venir) Trier nécessite $\Omega(n \log n)$ comparaisons.

Aujourd'hui

Calcul de x^n

Maximum

Complexité d'un problème

Problème du maximum

Trier

Pourquoi trier?

- Trier est une opération naturelle.
- Travailler sur des données triées est parfois plus efficace.
  - Exemple:
    - Rechercher un élément dans un tableau à n éléments: $O(n)$ (utiliser le principe de la fonction *Dans* précédente sur les listes).
    - Rechercher un élément dans un tableau trié à n éléments: $O(\log n)$ (par dichotomie).

Pourquoi trier?

- Trier est une opération naturelle.
- Travailler sur des données triées est parfois plus efficace.
  - Exemple:
    - Rechercher un élément dans un tableau à n éléments: $O(n)$ (utiliser le principe de la fonction *Dans* précédente sur les listes).
    - Rechercher un élément dans un tableau trié à n éléments: $O(\log n)$ (par dichotomie).
    - Trier devient intéressant dès que le nombre de recherches est en $\Omega(Complexite(tri)/n)$.