INF4705 — Analyse et conception d'algorithmes

CHAPITRE 10 : ALGORITHMES PROBABILISTES

Jusqu'ici, nous avons étudié des algorithmes déterministes : l'exemplaire à résoudre, ainsi que la description de l'algorithme, déterminent complètement la suite de calculs effectués, sans laisser place à des facteurs extérieurs comme le hasard.

Motivation : Les algorithmes probabilistes mettent le hasard à profit en faisant des choix aléatoires, ce qui a généralement des répercussions sur le temps de calcul.
Conséquence : Deux exécutions d'un algorithme probabiliste sur le même exemplaire peuvent prendre un temps différent et même donner des résultats différents.

On permettra entre autres, avec une faible probabilité :
- de retourner une mauvaise réponse ;
- de ne jamais terminer.

Un instant ! Revoyons notre définition d'un algorithme donnée au chapitre 1 :

> "Un algorithme est une suite finie d'instructions précises. Un algorithme qui résout un certain problème doit fonctionner correctement pour tous ses exemplaires."

Il s'agissait d'une définition d'algorithme *déterministe*, qu'il faut ici assouplir.
On peut résoudre plusieurs fois un exemplaire jusqu'à ce que l'exécution termine ou qu'on ait suffisamment confiance que notre solution est bonne.

Nous distinguons trois types (et demi) d'algorithme probabiliste :

algorithme	termine ?	correct ?
numérique probabiliste	√	
Monte Carlo	√	
Las Vegas		√
(Sherwood)	√	√

Exemple 1 *Les Olympiques*

Q : *En quelle année Montréal a-t-elle accueilli les Olympiques ?*

R (numérique probabiliste) : *entre 1971 et 1976, entre 1973 et 1978, entre 1975 et 1980,* entre 1970 et 1975*, entre 1972 et 1977.*

R (Monte Carlo) : *1976, 1976, 1976, 1976, 1977, 1976, 1976, 1976, 1642, 1976.*

R (Las Vegas) : *1976, 1976, 1976, Désolé !, 1976, ..., 1976,* `bus error`*, 1976, 1976.*

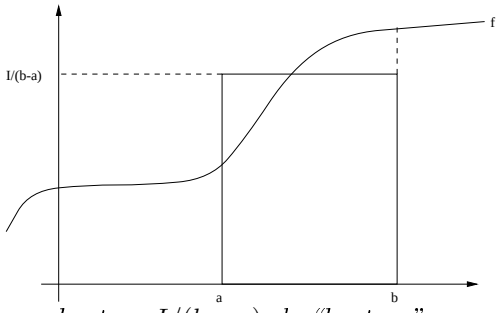
10.1 Algorithmes numériques probabilistes

Ils donnent une solution approximative (parfois sous la forme d'un intervalle de confiance) dont la précision augmente avec le temps de calcul alloué.
Ils sont généralement plus rapides qu'un algorithme exact mais parfois aussi une meilleure alternative si :
- les données expérimentales sont imprécises,
- la représentation interne des nombres est approximative (p.ex. irrationnels),
- la probabilité d'une erreur du matériel ("hardware") pendant l'exécution de l'algorithme exact est non négligeable.

Exemple 2 *Intégration numérique*
Voici le plus connu des algorithmes numériques probabilistes.

Soit $f : \mathbb{R} \to \mathbb{R}^{\geq 0}$ une fonction continue et $a, b \in \mathbb{R}$ tels que $a \leq b$.
L'aire sous la courbe $y = f(x)$ entre $x = a$ et $x = b$ est donnée par

$$I = \int_a^b f(x)dx.$$



Puisqu'un rectangle de même aire et même base a pour hauteur $I/(b-a)$, la "hauteur" moyenne de la courbe entre a et b sera également $I/(b-a)$.

Nous cherchons I pour une fonction pas facile à intégrer de façon symbolique. L'algorithme numérique probabiliste suivant estime cette hauteur moyenne par échantillonnage aléatoire puis multiplie le tout par $b-a$:

`inp`(f, a, b, n)
    $s \leftarrow 0$;
    pour $i \leftarrow 1$ **à** n **faire**
        $x \leftarrow$ *parmi* $[a, b]$ *selon distribution uniforme* ;
        $s \leftarrow s + f(x)$;
    retourner $(s/n) \times (b-a)$;

L'erreur sur la valeur estimée par cet algorithme est inversement proportionnelle à $\sqrt{n}$ (la preuve de cela dépasse le cadre du cours).

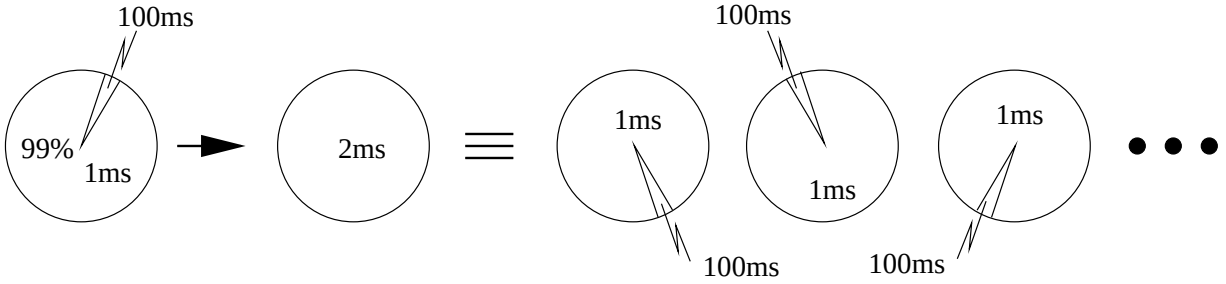
Une version déterministe de cet algorithme évaluant par exemple la fonction à intervalles réguliers obtiendrait d'aussi bons résultats avec moins de points d'évaluation, en une dimension.
L'algorithme probabiliste est par contre plus rentable pour évaluer des intégrales en dimension élevée (4 et plus), pour lesquelles la méthode déterministe systématique serait trop coûteuse.

Exemple 3 *Aiguille du comte de Buffon (18^{e}s.)*
Sur un plancher fait de lattes de bois de même largeur, laissons tomber des aiguilles dont la longueur est la moitié de la largeur des lattes : la probabilité qu'une aiguille chevauche deux lattes est égale à $\frac{1}{\pi}$.

On peut donc approximer π en prenant le rapport du nombre d'essais sur le nombre de succès.

10.2 Algorithmes Sherwood

Donnent toujours une (bonne) réponse mais le temps de calcul varie indépendamment de l'exemplaire, à cause des choix aléatoires qui sont faits.
utilité : Pour égaliser le temps de calcul espéré pour chacun des exemplaires, lorsqu'il y a une grosse différence entre le comportement moyen et en pire cas d'un algorithme déterministe.
C'est l'*Effet Robin Hood* : on redistribue l'inefficacité de quelques-uns à tout le monde.



Le comportement en pire cas existe toujours mais il n'est plus systématiquement associé à quelques exemplaires particuliers.

On rend ainsi un tel algorithme moins vulnérable aux mauvaises distributions d'exemplaires.

Exemple 4 *Tri de Hoare ("quicksort")*
$\mathcal{O}(n \lg n)$ en moyenne mais $\Omega(n^2)$ en pire cas (lorsque déjà trié).
D'autres algorithmes prennent $\mathcal{O}(n \lg n)$ en pire cas mais ont une plus grosse constante multiplicative.
Choisissons le pivot au hasard. $\Rightarrow \mathcal{O}(n \lg n)$ espéré en pire cas.

10.3 Algorithmes Las Vegas

La réponse donnée est toujours correcte mais, avec une probabilité faible, ils ne donnent pas de réponse du tout.
utilité : Pour l'efficacité obtenue en ne prenant pas le temps de trouver le bon choix.
Si la probabilité de succès d'un exemplaire x est $p(x)$, le nombre espéré de répétitions de l'algorithme Las Vegas afin d'obtenir une réponse est $1/p(x)$.

Soient $T_s(x)$ et $T_e(x)$ les temps d'exécution espérés pour un seul appel de l'algorithme en cas de succès et d'échec, respectivement. Le temps d'exécution espéré pour l'ensemble des répétitions sera

$$T_{tot}(x) = (\frac{1}{p(x)} - 1)T_e(x) + T_s(x).$$

Si $T_s(x) = T_e(x)$, on a simplement $T_{tot}(x) = T_s(x)/p(x)$.

Exemple 5 *Les n-reines*
L'ensemble des solutions à ce problème n'obéissent pas à une structure particulière. Nous pourrions vouloir générer plusieurs solutions différentes au problème.

Considérons l'exemplaire des 8-reines et trois algorithmes pour le résoudre. Le nombre de "noeuds explorés dans l'arbre" est une bonne estimation de leur efficacité relative : soient $NbN\text{œ}uds_s$ et $NbN\text{œ}uds_e$ le nombre de noeuds pour une exécution se terminant avec succès et échec, respectivement, et soit $NbN\text{œ}uds_{tot}$ le nombre de noeuds espéré lors d'une répétition de l'algorithme jusqu'à un succès.

approche	p	$NbN\text{œ}uds_s$	$NbN\text{œ}uds_e$	$NbN\text{œ}uds_{tot}$
déterministe (fouille à retour arrière bête)	*1.0000*	*114.00*	–	*114.00*
Las Vegas pur	*0.1293*	*9.00*	*6.97*	*55.93*
Las Vegas hybride 2-6 (meilleur hybride)	*0.8750*	*22.53*	*39.67*	*28.20*

En pratique, il y a un coût supplémentaire associé à la génération de nombres pseudo-aléatoires mais quand même...

39-reines :

approche	p	$NbN\text{œ}uds_{tot}$	T_{tot}
déterministe	*1.00*	10^{10}	*41 hrs*
Las Vegas hybride 29-10	*$\sim$0.21*	*500*	*8.5 ms*

10.4 Algorithmes Monte Carlo

Ils donnent une mauvaise réponse à l'occasion mais ils ont une probabilité élevée de retourner la bonne réponse et ce, *indépendamment de l'exemplaire considéré.*
Soit $p \in \mathbb{R}$ tel que $0 < p < 1$. Un algorithme de Monte Carlo est *p-correct* si la probabilité qu'il retourne la bonne réponse est au moins p, peu importe l'exemplaire : $P(\text{succès} \mid \text{exemplaire}) \geq p, \quad \forall$ exemplaire.
Un algorithme de Monte Carlo dont une des réponses possibles est nécessairement correcte lorsque obtenue est dit *biaisé*. Si au contraire la probabilité d'erreur est non nulle pour toutes les réponses possibles, on le dit *non biaisé* : $P(\text{succès} \mid \text{réponse}) < 1, \quad \forall$ réponse.

Exemple 6 *Vérification qu'un nombre est premier*
Nous parlerons du plus célèbre des algorithmes de Monte Carlo.

Étant donné n, un entier impair, est-il premier ?

Mis à part son intérêt en théorie des nombres, ce problème est très important en cryptologie. Il est assez difficile de le résoudre si une réponse exacte est exigée ($\mathcal{O}(\log^{12} n)$) ; le problème connexe de factorisation d'un nombre est probablement encore plus difficile.

Voici tout d'abord un algorithme déterministe datant de l'Antiquité, le Crible d'Ératosthène :

`Crible`*(n)*

Écrire dans l'ordre la liste de tous les nombres de 2 à n ;

pour $i \leftarrow 2$ **à** $\lfloor\sqrt{n}\rfloor$ **faire**

biffer tous les multiples de i (par sauts de i dans la liste) ;

tous les nombres non biffés sont premiers ;

La complexité de la consommation de ressources de cet algorithme, dans $\Omega(n)$, est malheureusement exponentielle dans la taille des données ($\lg n$) pour la même raison que l'algorithme de programmation dynamique pour le problème du sac à dos était pseudo-polynomial.

Voici un premier algorithme probabiliste, qui est $\frac{1}{2}$-correct :

`naif_non_biaisé`*(n)*

lancer une pièce de monnaie ;

si *pile* **alors retourner** *"premier" ;*

sinon retourner *"composé" ;*

Et un second :

`naif_biaisé`*(n)*

$a \leftarrow$ *choix aléatoire uniforme dans* $[2, \lfloor\sqrt{n}\rfloor]$ *;*

si a *divise* n **alors retourner** *"composé" ;*

sinon retourner *"premier" ;*

Le point de départ d'un algorithme probabiliste plus utile pour ce problème est un théorème de Fermat :

Théorème 1
n *premier* $\Rightarrow \forall\ 1 \leq a \leq n-1,\ a^{n-1}\ mod\ n = 1$

Sa contraposée,

$$\exists\ 1 \leq a \leq n-1,\ a^{n-1}\ mod\ n \neq 1 \Rightarrow n\ pas\ premier$$

mène à l'algorithme suivant :

`Fermat`*(n)*

$a \leftarrow$ *choix aléatoire uniforme dans* $[1, n-1]$ *;*

si $a^{n-1}\ mod\ n = 1$ **alors retourner** *"premier" ;*

sinon retourner *"composé" ;*

Prenant pour acquis que le calcul de l'élévation à une puissance en arithmétique modulaire peut s'effectuer en temps dans $\mathcal{O}(\lg^3 n)$, tout l'algorithme prend un temps dans $\mathcal{O}(\lg^3 n)$.

Mais que peut-on dire de la réponse ?

- *si l'algorithme retourne "composé" alors par le théorème n est certainement composé (sans qu'on ait pour autant exhibé ses facteurs) ; l'algorithme est donc biaisé*
- *si l'algorithme retourne "premier" alors ? ? ?*

Il est possible qu'on ait choisi par hasard un a qui satisfait l'équation ! C'est ce qu'on appelle des **faux témoins**. *Il en existe deux, triviaux, qu'on pourrait éliminer :* 1 *et* $n-1$. *Il en existe aussi d'autres comme par exemple* 4 *pour* $n = 15$ *puisque*

$$4^{14} \ mod\ 15 = 1.$$

Heureusement il y a en moyenne très peu de faux témoins : moins de 3.3% *des* a *pour les* $n < 1000$ *et encore moins que cela pour les* n *plus grands.*

Malheureusement certains n *possèdent un grand nombre de faux témoins : plus de* 50% *pour* 561 *et plus de* 99.9965% *pour 651693055693681 !*

En fait, `Fermat` *n'est* p*-correct pour aucun* p *puisqu'il est arbitrairement mauvais pour certains exemplaires.*

Un autre théorème nous vient en aide :

Théorème 2

n *premier* $\Rightarrow \forall\ 2 \leq a \leq n-2,\ a \in B(n)$

où $B(n) = \{a : [2 \leq a \leq n-2] \wedge [(a^t \ mod\ n = 1) \vee (\exists\ 0 \leq i < s,\ a^{2^i t} \ mod\ n = n-1)]\}$ *pour les entiers uniques* s *et* t *tels que* $n - 1 = 2^s t$ *avec* t *impair.*
Tout comme pour le théorème précédent, on peut se servir de sa contraposée pour dériver un algorithme :

`Miller-Rabin`*(*n*)*

  $a \leftarrow$ *choix aléatoire uniforme dans* $[2, n-2]$ *;*

  si `Btest`*(*a, n*)* **alors retourner** *"premier" ;*

  sinon retourner *"composé" ;*

La fonction suivante détermine si $a \in B(n)$ *:*

`Btest`*(*a, n*)*

  $s \leftarrow 0$ *;* $t \leftarrow n-1$ *;*

  répéter

    $s \leftarrow s+1$ *;* $t \leftarrow t \div 2$ *;*

  jusqu'à $t \ mod\ 2 = 1$

  $x \leftarrow a^t \ mod\ n$ *;*

  si $x = 1$ **ou** $x = n-1$ **alors retourner** *"vrai";*

  pour $i \leftarrow 1$ **à** $s-1$ **faire**

    $x \leftarrow x^2 \ mod\ n$ *;*

    si $x = n-1$ **alors retourner** *"vrai";*

  retourner *"faux";*

Étant donné un n *composé, tout* $a \in B(n)$ *est appelé* **faux témoin fort** *de* n. *Ceux-ci sont beaucoup moins nombreux que les faux témoins (ordinaires). Par exemple,* 4 *n'est pas un faux témoin fort de* 15 *puisque* $4^7 \ mod\ 15 = 4$.

A-t-on cette fois-ci des garanties quant à la probabilité de se tromper ? Oui, pour tout $n > 4$ *composé,* $|B(n)| \leq (n-9)/4$. *Ainsi lorsque* n *est*

- *premier, la probabilité que* `Miller-Rabin` *retourne la bonne réponse est* 1
- *composé, la probabilité que* `Miller-Rabin` *retourne la bonne réponse est au moins* $3/4$.

Nous avons donc un algorithme Monte-Carlo $3/4$*-correct pour déterminer si un nombre est premier.*

Et si nous répétions des appels indépendants à `Miller-Rabin` *?*

`Répète-Miller-Rabin`*(*n, k*)*

  pour $j \leftarrow 1$ **à** k **faire**
    si `Miller-Rabin`*(*n*)="composé"* **alors retourner** *"composé";*

retourner *"premier"* ;

Lorsque n est

– *premier, la probabilité que* `Miller-Rabin` *retourne la mauvaise réponse est* 0

– *composé, la probabilité que* `Miller-Rabin` *retourne la mauvaise réponse est au plus* $(1/4)^k$.

Nous obtenons donc un algorithme Monte-Carlo $(1-(1/4)^k)$*-correct. Son temps de calcul est dans* $\mathcal{O}(k \lg^3 n)$.

Pouvons-nous donc réduire la probabilité d'erreur de façon arbitraire ? Étant donné un ϵ *quelconque, choisissons* $k = \lceil \frac{1}{2} \lg 1/\epsilon \rceil$ *puisque*

$$1 - 4^{-k} \geq 1 - 2^{-\lg 1/\epsilon} = 1 - \epsilon$$

Il serait tout à fait possible d'appliquer cet algorithme à des entiers de quelques milliers de chiffres et avec une probabilité d'erreur inférieure à 10^{-100}.

En général, k répétitions d'un algorithme p-correct biaisé nous permettront d'obtenir un algorithme q-correct où

$$q = 1 - (1-p)^k$$

Ceci est un exemple d'**amplification de l'avantage stochastique**.
C'est particulièrement rentable pour les algorithmes biaisés mais on peut aussi l'appliquer aux algorithmes non biaisés en autant que leur avantage soit positif.
L'*avantage* d'un algorithme Monte Carlo p-correct est $p - 1/2$.
idée : On peut procéder ici aussi en répétant des appels indépendants à un algorithme de base mais on n'a pas de certificat d'authenticité d'une certaine réponse nous permettant de nous arrêter avec certitude. Par contre, on peut retourner la réponse majoritaire avec une certaine assurance.

`Répète-A`$(n, k$ /*impair*/$)$
  $c \leftarrow 0$;
  pour $j \leftarrow 1$ **à** k **faire**
    si A(n)="oui" **alors** c++ ;
    sinon c− ;
  si $c > 0$ **alors retourner** "oui" ;
  sinon retourner "non" ;

Exemple 7 *Soit un algorithme* 3/4*-correct non biaisé ; avantage :* 1/4 *; répétons 3 fois et retournons la réponse majoritaire.*
Énumérons les 8 cas possibles avec leur probabilité (B = bonne réponse ; M = mauvaise réponse) :

1er	*2e*	*3e*	*proba.*	*réponse*	
B	*B*	*B*	*27/64*	*B*	√
B	*B*	*M*	*9/64*	*B*	√
B	*M*	*B*	*9/64*	*B*	√
B	*M*	*M*	*3/64*	*M*	
M	*B*	*B*	*9/64*	*B*	√
M	*B*	*M*	*3/64*	*M*	
M	*M*	*B*	*3/64*	*M*	
M	*M*	*M*	*1/64*	*M*	

$27/64 + 9/64 + 9/64 + 9/64 = 27/32 > 84\% > 75\%$

En général, k répétitions d'un algorithme p-correct non biaisé à avantage positif nous permettront d'obtenir un algorithme q-correct où

$$q = \Sigma_{i=0}^{\lfloor k/2 \rfloor} \binom{k}{i} p^{k-i} (1-p)^i$$

Par exemple, si nous voulons utiliser un algorithme Monte Carlo non biaisé avec avantage 0.05 afin d'obtenir un algo 0.95-correct, il nous faut 269 répétitions. En contraste, un algorithme biaisé de même qualité (0.55-correct) n'aurait besoin que de 4 répétitions afin d'obtenir un algorithme 0.95-correct !