TCA 101, Informatique

Responsable et cours : Robert Strandh

Planning :

6 séances de cours
6 séances de TD
6 séances de TP
environ 3h de travail individuel par semaine

Web: http://dept-info.labri.fr/~strandh/Teaching/TCA101/2001-2002/Dir.html

Support de cours

Polycopié : *Introduction à l'Informatique* par Olivier Baudon

Transparents

Objectif et contenu

Objectif : Introduire des concepts de base en informatique

Contenu : Codage de l'information

Introduction à l'algorithmique

Internet (services)

Codage d'information

Un ordinateur n'est capable de traiter que des nombres.

Pour pouvoir stocker, transmettre et traiter des informations par un ordinateur, elles doivent donc être codées sous la forme de nombres.

La base la plus simple à utiliser pour les nombres est la base 2 (trou ou pas, ...).

Nous appelons un nombre codé en base 2 un nombre *binaire*.

Nombres

Soit b un entier naturel supérieur ou égal à 2 appelé la *base*. Tout entier naturel n peut s'écrire de façon unique sous la forme :

$$n = a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b^1 + a_0 b^0$$

avec $0 \leq a_i < b$. Les a_i sont les *chiffres* du nombre.

On écrit $n = (a_k a_{k-1} \ldots a_1 a_0)_b$, par exemple $20 = (20)_{10}$ ou $20 = (202)_3$ (car $2 \cdot 3^2 + 0 \cdot 3^1 + 2 \cdot 3^0 = 18 + 0 + 2 = 20$).

Schéma de Horner

Pour calculer n à partir des chiffres, on peut bien sûr utiliser la formule directement.

Mais c'est inefficace, car le calcul des puissances est coûteux.

On peut alors utiliser le Schéma de Horner :

$$a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b^1 + a_0 b^0$$

est la même chose que :

$$(\ldots (a_k b + a_{k-1}) b + \ldots + a_1) b + a_0$$

En informatique, on pense toujours à la manière dont on peut calculer quelque chose de façon efficace.

Schéma de Horner (suite)

Nous avons donc une méthode pour calculer un nombre en base b à partir des chiffres :

1. initialiser un accumulateur à 0,
2. s'il n'y a plus de chiffres, arrêter,
3. sinon multiplier l'accumulateur par la base et rajouter le premier chiffre du nombre,
4. enlever le premier chiffre du nombre,
5. répéter à partir de l'étape numéro 2.

Exemple du schéma de Horner

Exemple : nous souhaitons calculer la valeur de $(21021)_3$.

1. mettre 0 dans *acc*,
2. multiplier *acc* par 3 et rajouter 2 $\Rightarrow 0 \cdot 3 + 2 = 2$,
3. enlever le premier chiffre, $\Rightarrow$ 1021,
4. multiplier *acc* par 3 et rajouter 1, $\Rightarrow 2 \cdot 3 + 1 = 7$,
5. enlever le premier chiffre, $\Rightarrow$ 021,
6. multiplier *acc* par 3 et rajouter 0, $\Rightarrow 7 \cdot 3 + 0 = 21$,
7. enlever le premier chiffre, $\Rightarrow$ 21,
8. multiplier *acc* par 3 et rajouter 2, $\Rightarrow 21 \cdot 3 + 2 = 65$,
9. enlever le premier chiffre, $\Rightarrow$ 1,
10. multiplier *acc* par 3 et rajouter 1, $\Rightarrow 65 \cdot 3 + 1 = 196$,
11. enlever le premier chiffre, $\Rightarrow$ *rien*.
12. arrêter. Le résultat est 196.

Schéma de Horner (suite)

On peut utiliser le schéma de Horner à l'envers pour calculer les chiffres d'un nombre.

1. initialiser un accumulateur avec le nombre,
2. diviser l'accumulateur par la base et noter le reste,
3. si l'accumulateur contient 0, alors arrêter,
4. sinon, répéter à partir de 2.

Exemple du schéma de Horner

Exemple : nous souhaitons calculer les chiffres en base 4 du nombre 371.

1. mettre 371 dans *acc*,
2. diviser *acc* par 4 $\Rightarrow$ $371/4 = 92$ reste 3,
3. diviser *acc* par 4 $\Rightarrow$ $92/4 = 23$ reste 0,
4. diviser *acc* par 4 $\Rightarrow$ $23/4 = 5$ reste 3,
5. diviser *acc* par 4 $\Rightarrow$ $5/4 = 1$ reste 1,
6. diviser *acc* par 4 $\Rightarrow$ $1/4 = 0$ reste 1,
7. *acc* est 0, arrêter.

Donc $371 = (11303)_4$.

Multiplication et division par la base

Un nombre représenté en base b est facile à multiplier ou à diviser par la base.

Pour multiplier par la base, rajouter un 0 à la fin.

Pour diviser par la base, effacer le dernier chiffre (division entière).

Exemple de multiplication et division

Multiplier $(3452)_{10}$ par $(10)_{10}$ donne $(34520)_{10}$.

Multiplier $(1011010)_2$ par $(2)_{10} = (10)_2$ donne $(10110100)_2$.

Diviser $(12021)_3$ par $(3)_{10} = (10)_3$ donne $(1202)_3$.

Bit, octet, mot

Les nombres binaires sont particulièrement importants en informatique.

Un chiffre binaire s'appelle un *bit* de l'anglais "binary digit" (chiffre binaire).

Huit bits forment un *octet*. Le terme anglais est *byte*.

Un *mot* (anglais *word*) est une suite de bits particulièrement facile à traiter pour un ordinateur disponible. Sur un PC avec un processeur Intel Pentium, un mot contient 32 bits.

Le processeur d'un lave-linge a probablement des mots de 8 bits, et celui d'un console de jeux haut-de-gamme a des mots de 128 bits.

Nombre de valeurs possible

Un mot de n chiffres en base b peut prendre b^n valeurs différentes.

Exemple : Il y a exactement $10^4 = 10000$ codes possible pour une carte bancaire avec des codes de 4 chiffres décimaux.

Exemple : un mot binaire sur 4 bits peut prendre $2^4 = 16$ valeurs différentes.

Pour coder n valeurs différentes en base b, il faut $\lceil \log_b n \rceil$ chiffres.

$\lceil \log_b n \rceil$ est le plus petit entier k tel que $b^k \geq n$

Exemple : pour coder 11 valeur différentes en base 2, il faut $\lceil \log_2 11 \rceil = 4$ chiffres.

Ordre de grandeur

Il est relativement facile de convertir une puissance de 2 en un puissance de 10 approximative.

Il suffit de réaliser que $2^n = (10^{\log_{10} 2})^n = 10^{n \log_{10} 2} \approx 10^{0.3n}$

Par conséquent, $2^{10} \approx 10^{0.3 \cdot 10} = 10^3$.

Plus généralement : $2^{10n} \approx 10^{3n}$.

Habituellement, les puissances de 10 multiples de 3 sont appelées *kilo*, *méga*, *giga*, etc. En informatique, les même préfixes sont utilisés plutôt pour des puissances de 2 multiples de 10.

Un kilo-bit (ou kb) n'est donc pas mille bits, mais $2^{10} =$ 1024 bits. Un giga-octet (ou Go) est $2^{30} = 1073741824$ octets.

Entiers relatifs

Pour coder des entiers négatifs, il y a plusieurs possibilités.

Peut-être la façon la plus naturelle est de donner la valeur absolue avec un bit qui indique si le nombre est positif ou négatif.

Malheureusement, cette représentation nécessite un grand nombre de circuits pour la réalisation de l'addition avec un nombre négatif.

Pour cette raison, on utilise traditionnellement une représentation totalement différente que l'on appelle *complément à* 2.

Complément à 10 infini

Pour comprendre cette représentation, commençons par la base 10.

Imaginons un odomètre mécanique avec une infinité de roues. Lorsque une roue passe de 9 à 0, la roue à sa gauche avance d'une position. Lorsque une roue passe de 0 à 9, la roue à sa gauche recule d'une position.

Qu'est-ce qui arrive lorsque l'odomètre est en position $\ldots 000$ et si on recule d'une unité? Il passe à $\ldots 999$. C'est donc une représentation raisonnable du nombre -1.

Un nombre est donc positif s'il est précédé d'un nombre infini de 0 et négatif s'il est précédé d'un nombre infini de 9.

Addition

Supposons que nous ayons à notre disposition un circuit capable d'additionner deux nombres décimaux infinis positifs.

Ce circuit marche aussi avec les nombres négatifs en complément à 10.

Exemple : Pour additionner 234 (dont la représentation est ...000234) et -3 (dont la représentation est ...9997, on obtient

$$\begin{array}{cccccccc} . & . & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \\ . & . & 0 & 0 & 0 & 2 & 3 & 4 \\ . & . & 9 & 9 & 9 & 9 & 9 & 7 \\ \hline . & . & 0 & 0 & 0 & 2 & 3 & 1 \end{array}$$

ce qui donne la bonne réponse.

Soustraction

Pour calculer $x-y$, il faut en réalité calculer $x+(-y)$, ce qui nécessite une manière de calculer l'opposé d'un nombre.

Heureusement, c'est facile en représentation complément à 10 infinie. Il suffit de remplacer chaque chiffre c par $9-c$, puis de rajouter 1 au résultat.

Exemple : pour calculer l'opposé du nombre 4, dont la représentation est ...0004, on remplace d'abord chaque chiffre 0 par $9-0=9$ et le 4 par $9-4=5$, ce qui donne ...9995. Finalement on additionne 1. Le résultat est ...9996.

La raison que cela fonctionne est que $-x=(-1-x)+1$ et que l'opération qui remplace chaque chiffre c par $9-c$ est exactement le calcul $-1-x$ (la représentation de -1 est ...999).

Complément à 10 fini

Il est possible d'utiliser la représentation en complément à 10 avec un nombre de chiffres fini.

Pour la symétrie, il est préférable de pouvoir représenter autant de nombre négatifs que de nombres positifs (on considère ici que 0 est positif).

Pour cela, tous les nombre dont le premier chiffre est 0, 1, 2, 3 ou 4 sont considérés comme positifs et les autres comme négatifs.

Complément à 10 fini (suite)

Exemple d'addition en complément à 10 fini (4 chiffres de précision) :

Pour additionner 54 et -67, il faut d'abord convertir -67 en complément à 10. Cela donne $9999-0067+1 = 9932+1 = 9933$.

Puis pour l'addition, on obtient $0054 + 9933 = 9987$.

On peut vérifier que 9987 est la représentation de -13 en utilisant la même conversion : $9999-9987+1 = 0012+1 = 0013$.

Débordement

Avec une représentation finie, il y a toujours possibilité de *débordement*.

Le débordement arrive lorsque deux nombres positifs sont additionnés et le résultat semble négatif, ou lorsque deux nombre négatifs sont additionnés et le résultat semble positif.

Exemple : Nous souhaitons additionner 2894 et 3219. On obtient :

$$\begin{array}{rcccc} & 2 & 8 & 9 & 4 \\ + & 3 & 2 & 1 & 9 \\ \hline & 6 & 1 & 1 & 3 \end{array}$$

Mais un nombre qui commence par le chiffre 6 est négatif. Il y a donc débordement.

Complément à 2

Les techniques démontrées pour les nombres en complément à 10, marchent aussi pour le complément à 2, sauf que tout devient plus simple.

Un nombre dont le premier chiffre est 0 est considéré comme positif et un nombre qui commence par 1 est considéré comme négatif.

Pour calculer l'opposé d'un nombre, il suffit d'échanger 0 et 1, puis de additionner 1.

Exemple avec 8 bits de précision : l'opposé de 00110100 est $11001011 + 1 = 11001100$

Caractères

Pour coder du texte, il faut coder les caractères.

Solution simple : choisir un jeu de caractères de taille 2^n et coder un caractère avec n bits.

ASCII (American Standard Code for Information Interchange, ISO-646), 7 bits.

ISO-Latin-1 (ISO-8859-1), 8 bits.

Extensions du code ASCII

Avec ISO-Latin-1, les 128 premier caractères = ASCII.

Unicode (ISO-10646) code les caractères sur 16 bits. Les 256 premier caractères = ISO-Latin-1.

Son

Variation de la pression en fonction du temps.

Nécessite la *discrétisation*.

CD audio

La norme IEC 908.

Taux d'échantillonage : 44100 Hz.

Chaque échantillon est codé sur 16 bits.

Une heure de musique stéréo nécessite donc : $44100 \times 2 \times 2 \times 3600 = 635040000$ octets, soit environ 600Mo

MP3

Compression avec perte basée sur la psychoacoustique.

Permet des fichiers plus petits.

Codes auto-correcteurs

Permet de corriger un nombre d'erreurs fixe, par exemple un bit d'erreur sur un mot de 16 bits.

Il faut des bits supplémentaires, environ $\log n$ où n est le nombre de bits du mot pour corriger un bit d'erreur du mot.

Images

Même problème de discrétisation que pour le son.

Une image est divisée en *pixels*.

Chaque pixel a une couleur.

Une couleur est souvent codée sur trois octets (RGB, Red-Green-Blue).

HTML

HyperText Markup Language.

C'est le codage utilisé par le Web.

Permet de décrire des documents structurés contenant du texte, des images et des liens hypertextes.

HTML

Ce langage est basé sur un système de *balises*.

La plupart des balises fonctionnent comme des parenthèses étiquetées.

Structure générale d'une page :

```
<HTML>
<HEAD> ... </HEAD>
<BODY>
...
</BODY>
</HTML>
```

Exemple de HTML

```
<HTML>
<HEAD><TITLE>TCA-101, partie informatique</TITLE></HEAD>
<BODY>
<H1>TCA-101, partie informatique</H1>

<A HREF="../Common/Slides/slides.ps"> Les transparents en version
PostScript</A><P>

<A HREF="../Common/Slides/slides.pdf">
Les transparents en version PDF</A><P>
Pour le polycopié et les annales d'examens, veuillez consulter
<A HREF="http://dept-info.labri.u-bordeaux.fr/~baudon/TCA101/index.html
les pages web de M. Olivier Baudon</A>

</BODY>
</HTML>
```

Introduction à l'algorithmique

L'informatique est concerné par le traitement automatique d'information.

Cela nécessite :

- organiser l'information pour faciliter le traitement,
- inventer des méthodes efficaces pour le traitement,
- programmer un ordinateur pour exécuter ces méthodes.

Les deux premiers items font partie de la discipline de l'*algorithmique*.

Le dernier item fait partie de la discipline de la *programmation*.

[attention, la programmation couvre plus que ça]

Instructions

Un algorithme est généralement décrit comme une suite d'*instructions* à exécuter séquentiellement.

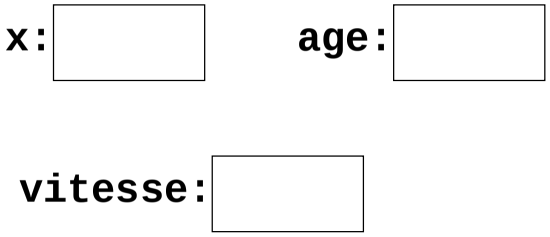
Ici nous ne considérons que les types d'instructions suivants :

- affectations,
- boucles,
- instructions conditionnelles.

Affectation

Une affectation donne une *valeur* à une *variable*.

Un variable est une boîte nommée dans la mémoire de l'ordinateur.



La valeur est calculée à partir d'une *expression*.

Exemple : `f(x, y) + 3 * age`

Affectation

L'affectation sera notée avec `:=`

Exemples :

```
coefficient := f(x, y) + 3 * age
```

```
x := x + 1
```

Lorsqu'une variable apparaît en partie droite d'une affectation, c'est sa valeur qui est employée.

Lorsqu'une variable apparaît en partie gauche d'une affectation, c'est sa référence (la boîte) qui est concernée.

Boucles

Lorsqu'une étape d'un algorithme doit être répétée plusieurs fois, il convient d'utiliser une *boucle*.

Il y existe plusieurs types de boucles. Ici nous allons utiliser deux types : `pour` et `tant que`.

Boucle `pour`

Soit v une variable et e_1, e_2 et e_3 trois expressions. La forme générale de la boucle `pour` est :

`for` v `from` e_1 `to` e_2 `by` e_3 `do`
   I_1;
   ...
   I_n;
`od;`

Boucle pour

Dans :

```
for v from e1 to e2 by e3 do
   I1;
   ...
   In;
od;
```

$I_1, I_2, \ldots I_n$ est une suite d'instructions, que l'on appelle le *corps* de la boucle.

Boucle pour

Dans :

```
for v from e1 to e2 by e3 do
   I1;
   ...
   In;
od;
```

Valeurs de v : e_1, $e_1 + e_3$, $e_1 + 2e_3$, ..., $e_1 + ke_3$ où k est le plus grand entier tel que $e_1 + ke_3 \leq e_2$ si la valeur de e_3 est positive, et $e_1 + ke_3 \geq e_2$ si la valeur de e_3 est négative.

Boucle pour

Dans :

```
for v from e1 to e2 by e3 do
   I1;
   ...
   In;
od;
```

e_3 est appelé le *pas*. Si on sais que le pas vaut 1, on peut simplifier la boucle :

```
for v from e1 to e2 do
   I1;
   ...
   In;
od;
```

Boucle pour

Dans :

for v **from** e_1 **to** e_2 **by** e_3 **do**
  I_1;
  ...
  I_n;
od;

v est appelé le *compteur* de boucle. Les noms usuels pour les compteurs de boucles sont `i`, `j` et `k`.

Exemple : puissance de deux

L'algorithme suivant calcule 2^n (où n est l'entier contenu dans la variable `n`) dans la variable `puiss2` :

```
puiss2 := 1;
for i from 1 to n do
  puiss2 := puiss2 * 2;
od;
```

Exemple : factorielle

L'algorithme suivant calcule la factorielle d'un entier contenu dans la variable `n` dans la variable `fact` :

```
fact := 1;
for i from 1 to n do
  fact := fact * i;
od;
```

Condition

Une *condition* est une *expression booléenne* (dont la valeur est *vrai* ou *faux*.

Exemples :

`x > 6`

`x <> 0 et f(x)`

Boucle `tant que`

Forme générale:

while c **do**
  I_1;
  ...
  I_n;
od;

où c est une condition.

Comme pour la boucle `pour`, I_1, ..., I_n est une suite d'instructions appelée le *corps* de la boucle qui sera répétée tant que la condition vaut *vrai*.

Exemple : nombres de bits

L'algorithme suivant compte dans la variable `nbits` le nombre de bits nécessaires pour coder l'entier contenu dans la variable `n`.

```
i := n / 2;
nbits := 1;
while i <> 0 do
  i := i / 2;
  nbits := nbits + 1;
od;
```

Le symbole <> signifie *différent de* et le symbole / désigne la division entière.

Transformation de boucles

Une boucle `pour` peut être exprimée sous la forme d'une boucle `tant que` de la manière suivante :

```
for v from e1 to e2 by e3 do
   I1;
   ...
   In;
od;
```

peut se réécrire en :

```
v := e1;
tant que v <= c2 faire
   I1;
   ...
   In;
   v := v + e3;
od;
```

Instructions conditionnelles

Deux formes :

```
if condition then
   I_1;
   ...
   I_n;
fi;
```

```
if condition then
   I_1;
   ...
   I_m;
else
   J_1;
   ...
   J_n;
fi;
```

Poids d'un entier

L'algorithme suivant compte dans une variable `poids` le nombre de bits à 1 d'un entier contenu dans la variable `n`.

```
i := n;
poids := 0;
while i <> 0 do
  if i % 2 = 1 then
    poids := poids + 1;
  fi;
  i := i / 2;
od;
```

Procédures

Une suite d'instructions *nommée* avec *paramètres*.

nom `:= proc(`*paramètres*`)`

`local` *variables locales*`;`

I_1`;`

...

I_m`;`

résultat;

`end;`

Exemple de procédure

```
puiss2 := proc(n)
  local p2, i;
  p2 := 1;
  for i from 1 to n do
    p2 := p2 * 2;
  od;
  p2;
end;
```

Exemple de procédure

```
fact := proc(n)
  local f, i;
  f := 1;
  for i from 1 to n do
    f := f * i;
  od;
  f;
end;
```

Fibonacci

$u_0 = 0$

$u_1 = 1$

$u_n = u_{n-1} + u_{n-2}$

Fibonacci

```
fibo := proc(n)
  local u, v, w, i;
  v := 0;
  u := 1;
  for i from 1 to n do
    w := u + v;
    v := u;
    u := w;
  od;
  v;
end;
```

Racine carrée

$u_0 = 1$

$u_n = \frac{u_{n-1} + x/u_{n-1}}{2}$

Racine carrée

```
rac2 := proc(x, epsilon)
  local u, v;
  u := 1.0;
  v := u + 2.0 * epsilon;
  while abs(u - v) > epsilon do
    v := u;
    u := (u + x / u) / 2.0;
  od;
  u;
end;
```

Point fixe d'une fonction

Un point fixe d'une fonction f est une valeur x telle que $f(x) = x$.

$u_0 = a$

$u_n = f(u_{n-1})$

La suite peut ne pas converger.

Point fixe d'une fonction

```
pf := proc(f, a, epsilon)
  local u, v;
  u := a;
  v := a + 2.0 * epsilon;
  while abs(u - v) > epsilon do
    v := u;
    u := f(u);
  od;
  u;
end;
```

Vecteurs

Un *vecteur* est une suite de variables indicées par des entiers.

En Maple, une telle structure s'appelle une *liste* (ce qui est dommage).

Constantes : suite de valeurs séparées par des virgules et encadrée par des crochets. Exemple : `[5,6,4,7,3]`.

Le nombre d'éléments du vecteur s'obtient grâce à la fonction `nops`. Exemple : `nops([5,6,4,7,3])` vaut 5.

Accès aux éléments

L'accès à l'élément numéro i d'un vecteur v se fait grâce à la notation $v[i]$, où v et i sont des expressions arbitraires (la valeur de v doit être un vecteur et la valeur de i doit être un entier).

Cet accès est très rapide. On dit $O(1)$, ce qui veut dire que le temps d'accès est indépendant du nombre d'éléments.

Rajouter ou supprimer un élément

En Maple, il y a une différence subtile entre *liste* et *séquence*.

Rajouter où supprimer un élément d'un vecteur est une opération lente. On dit $O(n)$ (où n est le nombre d'éléments de vecteur), ce qui veut dire que le temps est proportionnel au nombre d'éléments.

Recherche d'un élément

Exemple : un vecteur de tous les étudiants de TCA101. La recherche détermine si une personne est en fait étudiant de TCA101.

```
present := proc(x, l)
  local i, p;
  p := false;
  for i from 1 to nops(l) do
    if l[i] = x then
      p := true;
    fi;
  od;
  p;
end;
```

Le temps d'exécution est $O(n)$ au pire.

Simple amélioration

```
present := proc(x, l)
  local i, p;
  p := false;
  i := 1;
  while i <= nops(l) and not p do
    if l[i] = x then
      p := true;
    fi;
    i = i + 1;
  od;
  p;
end;
```

Élément maximum

```
max := proc(l)
  local i, m;
  m := l[1];
  for i from 2 to nops(l) do
    if l[i] > m then
      m := l[i];
    fi;
  od;
  m;
end;
```

Indice de l'élément maximum

```
max := proc(l)
  local i, im;
  im := 1;
  for i from 2 to nops(l) do
    if l[i] > l[im] then
      im := i;
    fi;
  od;
  im;
end;
```

Moyenne

```
moyenne := proc(l)
  local i, s;
  s := 0;
  for i from 1 to nops(l) do
    s := s + l[i];
  od;
  s / nops(l);
end;
```