Algorithmique pour l'Algèbre Linéaire

Thomas Fernique
CNRS & Univ. Paris 13

L1 Maths-Info
27 janvier 2017

Menu

Menu

Contenu

Deux blocs intégrés dans le cours d'algèbre linéaire.

Semaines 3–5 (maintenant) : manipuler des matrices en C.
Semaines 9–11 (fin mars) : quelques techniques d'algorithmique.

Chaque bloc :

- un cours (1h30)
- deux TD (1h30+1h30)
- deux TP (1h30+3h00)

Philosophie :

- Autre point de vue sur le cours d'algèbre ;
- initiation à l'algorithmique.

Evaluation

Contrôle continu : selon le travail fourni pendant les TD et TP.

Partiels : une question intégrée à chacun des deux partiels.

Coefficients : géré par Thomas Duyckaerts.

Menu

Variables

Déclaration : nom et type.
Affectation : immédiate ou retardée.

```
int i=1;
float r;
r=3.14159256;
double z=0.10010100100101001010010010;
char a='b';
```

Toute variable doit obligatoirement être déclarée !

Tests

Structure `if...else` (si...sinon) avec un test :

```
if (i==0)
 {...}
else
 {...}
```

Opérateurs logiques classiques :

```
if ((i!=0 && j>10) || k<3)
 {...}
```

Boucles

Si l'on sait à l'avance combien de fois on itère :

```
int j,k=0;
for (j=0;j<100;j++)
{
 k=k+j;
}
```

Si le nombre d'itérations est conditionnel :

```
int i=101;
while (i>100)
{
 i=rand();
}
```

Fonctions

Déclaration (facultative si utilisée seulement après que le code a été donné) :

```
int somme(int,int);
```

Code :

```
int somme(int a,int b)
{
 return a+b;
}
```

Récursivité :

```
int blup(int n)
{
 if (n==1) return 1;
 else return n*blup(n-1);
}
```

Pointeurs

Mémoire d'un ordinateur $\simeq$ rue où sont rangées les variables.

Adresse d'une variable `t` : `&t` (type entier).
Déclaration d'une adresse `a` : selon le type de ce qu'elle pointe.
Contenu à l'adresse `a` : `*a` (type selon la déclaration de `a`)

```
int i=5;
int* a=&i;
int j0=*a;
int j1=*(a+1);
int j2=a[2];
```

Écrire n'importe où dans la mémoire $\rightsquigarrow$ `segmentation fault`.

Menu

Tableau statique à une entrée

Tableau stocké dans le code même du programme.

```
int r[10];
int s[]={1,2,3};
int i=s[0];
s[1]=2;
int j=r[0]+s[2];
```

Attention : le premier indice est 0 !

Exemple

Que fait cette fonction ?

```
int somme(int t[],int n)
{
 int i;
 int s=0;
 for(i=0;i<n;i++)
 {
  s=s+t[i];
 }
 return s;
}
```

Tableau statique à plusieurs entrées

Tableau toujours stocké dans le code même du programme.

```
int r[3][3];
int s[][]={{1,8},{4,8}};
int i=r[0][1]+s[1][0];
float t[2][2][2];
```

Deux entrées : idéal pour les matrices !

Un exemple

Que fait cette fonction ?

```
int somme(int t[][],int p, int q)
{
 int i,j;
 int s=0;
 for(i=0;i<p;i++)
  for(j=0;j<q;j++)
   s=s+t[i][j];
 return s;
}
```

Un exemple

Que fait cette fonction ?

```
int somme(int t[][],int p, int q)
{
 int i,j;
 int s=0;
 for(i=0;i<p;i++)
  for(j=0;j<q;j++)
   s=s+t[i][j];
 return s;
}
```

Erreur de compilation : “array type has incomplete element type” !

Le problème

La mémoire d'un ordinateur est intrinsèquement unidimensionnelle.

```
int somme(int t[],int p, int q)
{
 int i,j;
 int s=0;
 for(i=0;i<p;i++)
  for(j=0;j<q;j++)
   s=s+t[i*q+j];
 return s;
}
```

Il faut lui expliquer comment y "replier" un tableau à deux entrées.

Un autre problème

La taille des tableaux doit être connue à la compilation.

Ceci interdit beaucoup de choses pourtant naturelles :

```
int i=10;
int r[i];
int j=gros_calcul();
int s[j];
int t[]=initialise(10);
int t[10]=initialise();
```

Allocation dynamique

Pour y remédier, on demande de la mémoire *pendant l'exécution* :

```
int* r=malloc(10*sizeof(int));
r[4]=0;
int i=r[9];
int j=5;
int* s=malloc(j*sizeof(int));
int* t=initialise(j);
free(t);
free(s);
free(r);
```

On n'oublie pas de rendre à la fin la mémoire allouée, et on n'écrit pas en dehors de cette mémoire (sinon `segmentation fault`).

Menu

Algorithme et complexité

Algorithme $\simeq$ recette de cuisine.

```
void omelette(int nb_oeufs)
{
 int i;
 for (i=0;i<nb_oeufs;i++)
  ajouter_oeuf();
 melanger();
 cuire();
 printf("à table !\n");
}
```

Complexité : comment varie le temps en fonction des quantités ?

Exemple 1 : multiplication d'entiers

39.4μs pour le produit suivant (résultat sur 4 bits) :

$$3 \times 5$$

42μs pour celui-ci (résultat sur 32 bits) :

$$65521 \times 65537$$

47μs pour celui-ci (résultat sur 256 bits) :

$$340282366920938463463374607431768211297 \times 340282366920938463463374607431768211507$$

Exemple 2 : factorisation d'entier

764μs pour factoriser 15 en

$$3 \times 5$$

1.47 ms pour factoriser 4294049777 en

$$65521 \times 65537$$

7 min. pour (re)factoriser le produit

$$340282366920938463463374607431768211297$$
$$\times$$
$$340282366920938463463374607431768211507$$

Comparaison

Considérons des algorithmes dont le nombre $c(n)$ d'opérations effectuées en fonction de la taille n de l'entrée est respectivement

$$\begin{aligned} c_1(n) &= 2017n + 1664 \\ c_2(n) &= 1664n + 2017 \\ c_3(n) &= 4n^2 \log(n) + 416 \\ c_4(n) &= 32n^3 - 52n^2 \\ c_5(n) &= 8.6^n + 0.33n^5 \end{aligned}$$

Lequel est le plus rapide ?

Complexité asymptotique : notation O

On note $O(g)$ les fonctions f telles que f/g est bornée.
Ce sont celles qui ne croissent asymptotiquement pas plus vite.

$$\begin{array}{lcll} c_1(n) & = & 2017n + 1664 & O(n) \\ c_2(n) & = & 1664n + 2017 & O(n) \\ c_3(n) & = & 4n^2 \log(n) + 416 & O(n^2 \log(n)) \\ c_4(n) & = & 32n^3 - 52n^2 & O(n^3) \\ c_5(n) & = & 8.6^n + 0.33n^5 & O(8.6^n) \end{array}$$

Choix de g : compromis entre simplicité et optimalité.

Complexité asymptotique : notation Θ

On note $\Theta(g)$ les fonctions f telles que f/g et g/f sont bornées.
Ce sont celles qui croissent asymptotiquement comme g.

$$\begin{array}{lcll} c_1(n) & = & 2017n + 1664 & \Theta(n) \\ c_2(n) & = & 1664n + 2017 & \Theta(n) \\ c_3(n) & = & 4n^2 \log(n) + 416 & \Theta(n^2 \log(n)) \\ c_4(n) & = & 32n^3 - 52n^2 & \Theta(n^3) \\ c_5(n) & = & 8.6^n + 0.33n^5 & \Theta(8.6^n) \end{array}$$

Le choix de g est à constante multiplicative près.

Retour sur multiplication et factorisation

Rappel : le temps de multiplication semblait croître beaucoup plus vite avec la taille des nombres que le temps de factorisation.

Retour sur multiplication et factorisation

Rappel : le temps de multiplication semblait croître beaucoup plus vite avec la taille des nombres que le temps de factorisation.

Multiplication :

- Algorithme ?
- Complexité ?
- Peut-on faire mieux ?

Retour sur multiplication et factorisation

Rappel : le temps de multiplication semblait croître beaucoup plus vite avec la taille des nombres que le temps de factorisation.

Multiplication :

- Algorithme ?
- Complexité ?
- Peut-on faire mieux ?

Factorisation :

- Algorithme ?
- Complexité ?
- Peut-on faire mieux ?

Un dernier exemple

Alice choisit un nombre entre 1 et n que Bob doit deviner.
À chaque essai de Bob, Alice dit “plus grand” ou “plus petit”.

Quel algorithme pour Bob ? Quelle complexité ?

Ayez le polycopié de cours en TD/TP !

Il est disponible sur la page de Thomas Duyckaerts :
`www.math.univ-paris13.fr/~duyckaer/enseignement.html`

Et ces slides sur la page de Thomas Fernique :
`lipn.univ-paris13.fr/~fernique/teaching.html`