Programmation et algorithmique

IN101

ENSTA Paris - TC 1ère année

François Pessaux

U2IS

2022-2023

prenom.nom@ensta-paristech.fr

Passer à des programmes plus compliqués

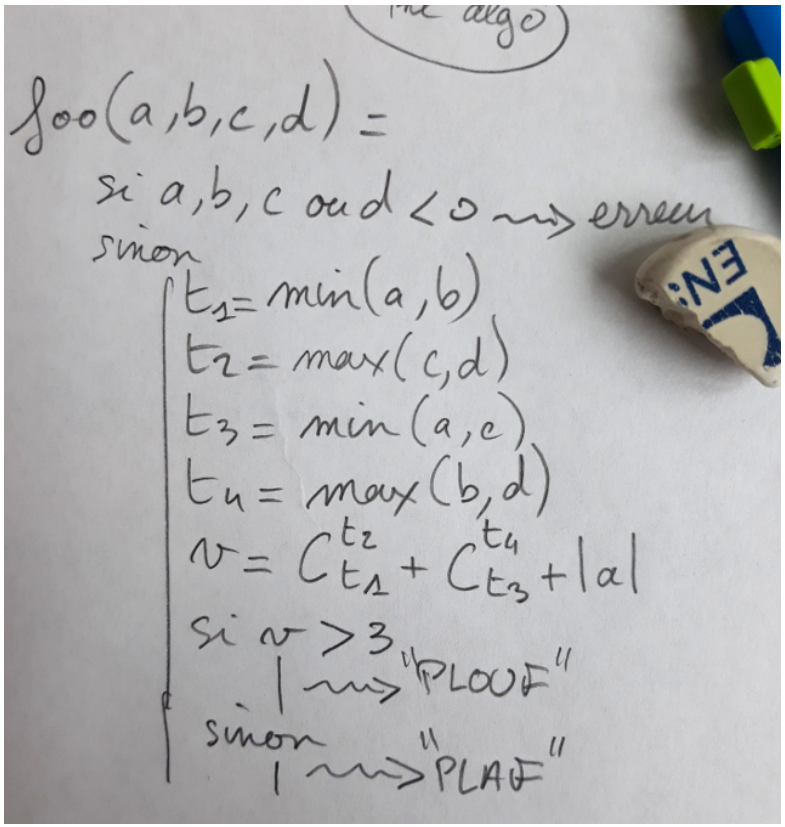
Vu dans le cours précédent. . .

- Modéliser le problème à résoudre.
- Réfléchir aux domaines des entrées et des sorties.
- Décomposer en sous-problèmes.
- User et abuser de papier, crayon, dessins, exemples. . .
- . . . . . .
- Rédiger dans un langage de programmation.

Structurer la rédaction du code

- Par étapes successives, on a raffiné l'algorithme.
- Par étapes successives, on va rédiger le programme.

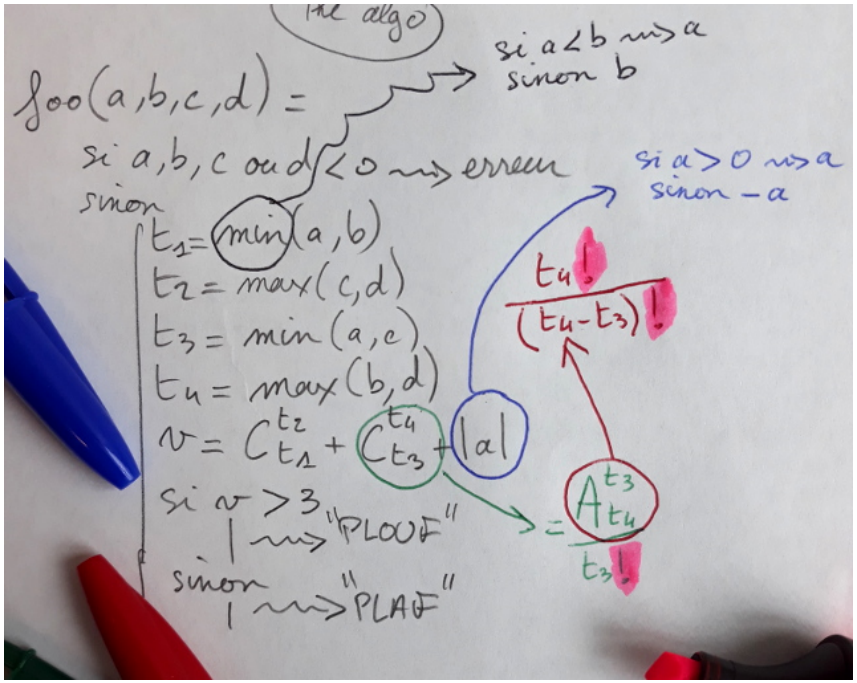
- Ça peut faire beaucoup de code à écrire.
- Ça peut nécessiter de découper en « unités de calcul » séparées.



La meilleure solution pour ne plus rien y comprendre

```
void foo (int a, int b, int c, int d)
      {
  int i, t1, t2, t3, t4, fact_t1,
       fact_t1_t2, arr_t1_t2, abs_a ;
  int comb_t1_t2, fact_t3, fact_t3_t4
       , arr_t3_t4, comb_t3_t4 ;
  if ((a < 0) || (b < 0) || (c < 0)
       || (d < 0))
    printf ("erreur") ;
  if (a < b) t1 = a ;
  else t1 = b ;
  if (c > d) t2 = c ;
  else t2 = d ;
  if (a < c) t3 = a ;
  else t3 = c ;
  if (b > d) t4 = b ;
  else t4 = d ;
  fact_t1 = 1 ;
  for (i = 1; i <= t1; i++)
    fact_t1 = fact_t1 * i ;
  fact_t1_t2 = 1 ;
  for (i = 1; i <= t1 - t2; i++)
    fact_t1_t2 = fact_t1_t2 * i ;
  arr_t1_t2 = fact_t1 / fact_t1_t2 ;
  comb_t1_t2 = arr_t1_t2 / t2 ;
  fact_t3 = 1 ;
  for (i = 1; i <= t3; i++)
    fact_t1 = fact_t3 * i ;
  fact_t3_t4 = 1 ;
  for (i = 1; i <= t3 - t4; i++)
    fact_t3_t4 = fact_t3_t4 * i ;
  arr_t3_t4 = fact_t3 / fact_t3_t4 ;
  comb_t3_t4 = arr_t3_t4 / t4 ;
  if (a < 0) abs_a = -a ;
  else abs_a = a ;
  if (comb_t1_t2 + comb_t3_t4 + abs_a
       > 3)
    printf ("PLOUF\n") ;
  printf ("PLAF\n") ;
}
```

Un peu de structure et de réutilisation



- Calculs décomposables en sous-calculs (problème / sous-problème).
- C_n^k se calcule à partir de A_n^k.
- C_n^k et A_n^k se calculent à partir de $n!$.

- ⇒ Décomposer en plusieurs fonctions.
- Gain de lisibilité et maintenabilité : nommage du calcul.
- Possibilité de réutiliser sans dupliquer le code ⇒ réutilisabilité
  - $n!$, `min`, `max` utilisés plusieurs fois.

Le code une fois structuré

```
int fact (int n) {
  int res = 1 ;
  for (int i = 1; i <= n; i++)
    res = res * i ;
  return res ;
}

int argt (int n, int k) {
  return fact (n) / fact (n - k) ;
}

int comb (int n, int k) {
  return argt (n, k) / fact (k) ;
}

int min (int a, int b) {
  if (a < b) return a ;
  return b ;
}

int max (int a, int b) {
  if (a > b) return a ;
  return b ;
}

int abs (int x) {
  if (x < 0) return -x ;
return x ;
}
```

```
void foo (a, b, c, d) {
  if ((a < 0) || (b < 0) || (c < 0)
      || (d < 0))
    printf ("erreur\n") ;
  int t1 = min (a, b) ;
  int t2 = max (c, d) ;
  int t3 = min (a, c) ;
  int t4 = max (b, d) ;
  if (comb (t1, t2) + comb (t3, t4) +
      abs (a) > 3)
    printf ("PLOUF") ;
  printf ("PLAF") ;
}
```

- Plus court.
- Plus lisible.
- Plus réutilisable.
- ⇒ Plus maintenable
  - corrections,
  - évolutions. . .

Beaucoup de code à écrire (c.f. diapo 4)

- Première solution : écrire au Km, lancer à la fin.

```
$ gcc -c -Wall junk.c
junk.c:10:14: error: use of undeclared identifier 't2'
  if (c > d) t2 = c ;
junk.c:11:8: error: use of undeclared identifier 't2'
  else t2 = d ;
junk.c:12:18: error: expected expression
  if (a < c) t3 := a
junk.c:20:25: error: use of undeclared identifier 't2'
  for (i = 1; i <= t1 - t2; i++)
junk.c:27:3: error: use of undeclared identifier 'fact_t3_t4'; did you mean
      'fact_t1_t2'?
junk.c:4:31: note: 'fact_t1_t2' declared here
  int i, t1, t3, t4, fact_t1, fact_t1_t2, arr_t1_t2, abs_a ;
junk.c:29:5: error: use of undeclared identifier 'fact_t3_t4'; did you mean
      'fact_t1_t2'?
...
... 459 autres à venir...
```

- Décourageant. . .

Développement incrémental : en mathématiques

- Soit à prouver $\forall f\, g\, x,\ x > 0 \wedge f(x) > 0 \wedge g(x) > 0 \Rightarrow f(2x) + g(2x) > 0$
- Démonstration construite par étapes.
  - $H_0 : x > 0$
  - $H_1 : f(x) > 0$
  - $H_2 : g(x) > 0$
  - Montrons que $f(2x) + g(2x) > 0$ par :
    - ★ preuve que $f(2x) > 0$
    - ★ preuve que $g(2x) > 0$
    - ★ CQFD par propriété de $+$
- Sous-buts ($f(2x) > 0$ et $g(2x) > 0$) non encore prouvés...
- ...mais vérification de la cohérence globale possible.
- Étapes suivantes : prouver les sous-buts.
  - Montrons que $f(2x) > 0$ par :
  - ...
- Construction incrémentale.

Développement incrémental : en informatique

- Écrire son (gros) programme par étapes.
- Vérifier la correction syntaxique du code au fur et à mesure.
  - En cours d'écriture d'une longue fonction.
  - En cours d'écriture d'un long programme (plusieurs fonctions).
- Vérifier correction d'exécution du code au fur et à mesure.
  - Tests : cf. diapos ≥ 16.
  - Vérifier les fonctions déjà écrites.
  - Vérifier les fonctions partiellement écrites
    - partie déjà écrite OK ?
  - Remplacer les fonctions non écrites par des fonctions triviales
    - retourne une valeur constante,
    - retourne une valeur aléatoire dans un domaine choisi...

Défaults courants de conception d'algorithme

Le syndrome affichage

« Écrire une fonction qui calcule la factorielle de son argument. »

```
int fact (int n) {
  int res = 1 ;
  for (int i = 1 ;i <= n; i++)
    res = res * i ;
  printf ("%d\n", res);
}
```

- Utilisons la fonction pour faire autre chose que calculer 5!

```
int main () {
  printf ("%d\n", fact (5) * 2) ;
  return 0 ;
}
```

```
$ gcc -Wall foo.c
foo.c:8:1: warning: control reaches end of non-void function [-Wreturn-type]
1 warning generated.
$ ./a.out
120
0
```

- Afficher n'est pas retourner une valeur. Ne sert qu'à l'utilisateur.
- Sauf spécification contraire : un programme affiche son résultat, une fonction retourne son résultat.

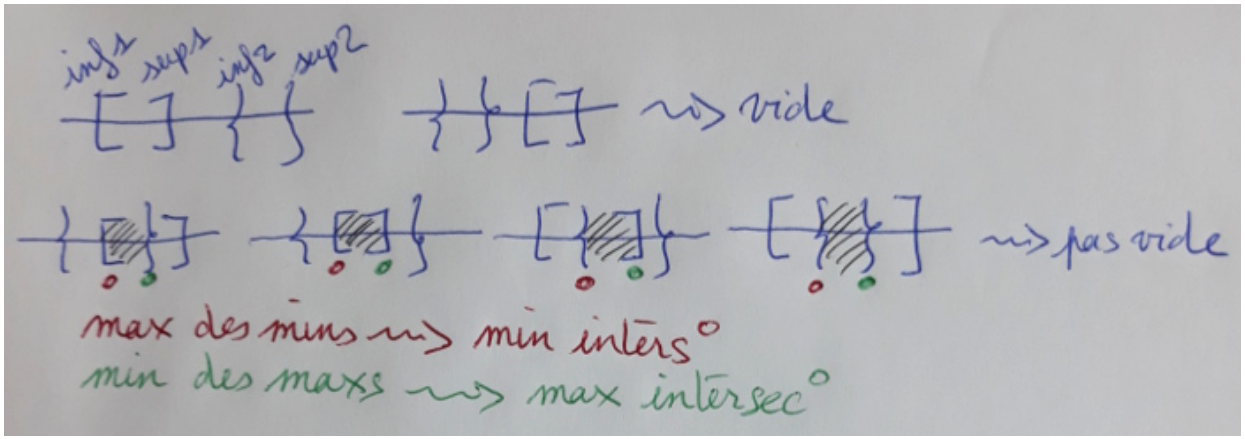
Syndromes duplication, imbrication, fouillis dans le code

```
void inter_interv (int inf1, int sup1, int inf2, int sup2) {
  if (inf1 < inf2) {
    if (sup1 < inf2) { printf ("[]\n") ; return ; }
  }
  if (inf2 < inf1) {
    if (sup2 < inf1) { printf ("[]\n") ; return ; }
  }
  if (inf1 <= inf2) {
    if (sup1 >= inf2) {
      if (sup1 >= sup2) { printf ("[%d; %d]\n", inf2, sup2) ; return ; }
      else { printf ("[%d; %d]\n", inf2, sup1) ; return ; }
    }
  }
  if (inf2 <= inf1) {
    if (sup2 >= inf1) {
      if (sup2 >= sup1) { printf ("[%d; %d]\n", inf1, sup1) ; return ; }
      else { printf ("[%d; %d]\n", inf1, sup2) ; return ; }
    }
  }
}
```

- Répétition de mêmes structures de code.
- Imbrication importante des structures de contrôle.
- Tests couvrent tous les cas ? Se recouvrent ?

Syndromes duplication, imbrication etc. (2)

- Réflexion et décomposition en sous-problèmes non satisfaisantes.
- ⇒ Besoin d'une vue plus synthétique du problème global.




```
int min (int a, int b) { return ((a < b) ? a : b) ; }
int max (int a, int b) { return ((a > b) ? a : b) ; }

void inter_interv (int inf1, int sup1, int inf2, int sup2) {
  if ((sup1 < inf2) || (sup2 < inf1)) printf ("[]\n") ;
  else {
    int inf = max (inf1, inf2) ;
    int sup = min (sup1, sup2) ;
    printf ("[%d; %d]\n", inf, sup) ;
  }
}
```

Tester son programme

Pourquoi tester ?

- « Parfois », ça ne marche pas. . .
- « Juste » besoin de s'en apercevoir puis de corriger.

- L'algorithme peut être erroné.
- L'implantation (programme) peut être erroné malgré un algorithme correct.
- Les deux peuvent être erronés.

- Pas maintenant un cours de test logiciel (2A filière STIC).
- Juste quelques conseils informels pour guider l'intuition.

Comment tester ?

- En exécutant son programme : tests haut-niveau.
- En exécutant sa (ses) fonctions : tests bas-niveau.
- Dans tous les cas : quelles valeurs des entrées utiliser ?
- Ne pas vérifier la conformité des résultats à partir de ce que l'on lit dans le code.

Tester à partir du code

« *Écrire une fonction qui retourne le min de ses 2 arguments.* »

```
int min (int x, int y) {
  if (x < y) return y ;
  return (x) ;
}
```

- Dans le code on voit : 2 entrées.
- Dans le code on voit : 1 conditionnelle ⇒ 2 cas.
- « Testons avec 5 et 6. Comme 5 < 6, ça doit retourner 6 ».

```
----> min (5, 6)
6
```

- « *Ça marche, on a bien* 6 *à l'exécution* ».
- On a juste vérifié que le compilateur n'est pas buggé.
- On a juste vérifié que ça exécute bien ce qui est écrit.
- On n'a rien vérifié par rapport à l'attendu initial (spécification).

Tester à partir de la spécification

1. Trouver des valeurs d'entrée.
2. Vérifier que résultat est conforme à l'attendu et non au code.

- Tests fonctionnels (sans regarder le code)
  - Valeurs devant donner ok (cas nominaux).
  - Valeurs devant donner erreur (cas d'erreurs).
  - Valeurs aux limites de ok et erreur.
- Tests structurels (en regardant le code)
  - Valeurs permettant d'activer chaque branche du programme.
    - Conditionnelle : 2 branches.
    - Boucle : 0 itération, n itérations.
  - Valeurs de sous-domaines d'opérations utilisées dans le programme.

Tests fonctionnels (1/5)

« Trouver le plus petit des entiers dans un fichier texte dont le nom est passé en argument de la ligne de commande. »

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
  int i, the_min ;
  FILE *file = fopen (argv[1], "rb") ;

  while (!feof (file)) {
    fscanf (file, "%d", &i) ;
    if (i < the_min) the_min = i ;
    fscanf (file, "%d", &i) ;
  }
  fclose (file) ;
  printf ("Min is %d\n", the_min) ;
  return 0 ;
}
```

- Sans nom de fichier :

```
$ gcc -Wall minfile_broken.c
$ ./a.out
Segmentation fault: 11
```

Tests fonctionnels (2/5)

« Trouver le plus petit des entiers dans un fichier texte dont le nom est passé en argument de la ligne de commande. »

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
  int i, the_min ;
  FILE *file = fopen (argv[1], "rb") ;

  while (!feof (file)) {
    fscanf (file, "%d", &i) ;
    if (i < the_min) the_min = i ;
    fscanf (file, "%d", &i) ;
  }
  fclose (file) ;
  printf ("Min is %d\n", the_min) ;
  return 0 ;
}
```

- Fichier non accessible :

```
$ ./a.out /tmp/doesntexist
Segmentation fault: 11
```

Tests fonctionnels (3/5)

« Trouver le plus petit des entiers dans un fichier texte dont le nom est passé en argument de la ligne de commande. »

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
  int i, the_min ;
  FILE *file = fopen (argv[1], "rb") ;

  while (!feof (file)) {
    fscanf (file, "%d", &i) ;
    if (i < the_min) the_min = i ;
    fscanf (file, "%d", &i) ;
  }
  fclose (file) ;
  printf ("Min is %d\n", the_min) ;
  return 0 ;
}
```

- Fichier vide :

```
$ ll /tmp/empty
-rw-r--r--  1 didou  wheel  0 12 mar 13:20 /tmp/empty
$ ./a.out /tmp/empty
Min is 0
```

Tests fonctionnels (4/5)

« Trouver le plus petit des entiers dans un fichier texte dont le nom est passé en argument de la ligne de commande. »

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
  int i, the_min ;
  FILE *file = fopen (argv[1], "rb") ;

  while (!feof (file)) {
    fscanf (file, "%d", &i) ;
    if (i < the_min) the_min = i ;
    fscanf (file, "%d", &i) ;
  }
  fclose (file) ;
  printf ("Min is %d\n", the_min) ;
  return 0 ;
}
```

/tmp/data1.txt

```
3
5
67
34
56
8
-90
-45
45
```

- Fichier bien formé :

```
$ ./a.out /tmp/data1.txt
Min is  -90
```

Tests fonctionnels (5/5)

« Trouver le plus petit des entiers dans un fichier texte dont le nom est passé en argument de la ligne de commande. »

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
  int i, the_min ;
  FILE *file = fopen (argv[1], "rb") ;

  while (!feof (file)) {
    fscanf (file, "%d", &i) ;
    if (i < the_min) the_min = i ;
    fscanf (file, "%d", &i) ;
  }
  fclose (file) ;
  printf ("Min is %d\n", the_min) ;
  return 0 ;
}
```

/tmp/data2.txt

```
3
-500
67
34
56
8
-90
-45
45
```

- Fichier bien formé :

```
$ ./a.out /tmp/data2.txt
Min is  -90
```

Résultat des tests fonctionnels

- 4 tests sur 5 en échec.
- Corriger le programme pour le rendre correct.
- Corriger le programme pour le rendre plus robuste.

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
  int i, the_min ;
  if (argc != 2) {    /* Check the presence of the command line argument. */
    printf ("Error. Wrong number of arguments.\n") ;
    return 1 ;
  }
  FILE *file = fopen (argv[1], "rb") ;
  if (file == NULL) {    /* Check if the file is opened. */
    printf ("Error. File not readable.\n") ;
    return 1 ;
  }
  if (fscanf (file, "%d", &i) == EOF) {    /* Does it contain data ? */
    printf ("Error. Empty file.\n") ;
    fclose (file) ;    /* Don't forget to close the opened file. */
    return 1 ;
  }
  while (!feof (file)) {    /* Read until the end of file. */
    if (i < the_min) the_min = i ;
    fscanf (file, "%d", &i) ;    /* Try to get the next data. */
  }
  fclose (file) ;
  printf ("Min is %d\n", the_min) ;
  return 0 ;
}
```

Tests fonctionnels vs structurels

- Explorer toutes les entrées possibles : impossible.
- Tests fonctionnels précédents : fichiers bien formés choisis aléatoirement...
- ... ou presque.
- Second fichier avec *min* en second : choisi en ayant regardé le code.
- Trouver ce cas sans regarder le code : beaucoup plus dur.

⇒ Combiner les 2 types de tests.

Tests structurels (1/3)

« *Calculer le nombre de chiffres d'un nombre écrit en base 10.* »

```
int ndigits (int n) {
  return (1 + (int) log10 (n)) ;
}
```

- Dans le code : $\log_{10}$.
- Définie uniquement sur $\mathbb{R}^{*+}$.
- Idée de test : -100.

```
$ ./a.out -100
-2147483647
```

Tests structurels (2/3)

« *Calculer le nombre de chiffres d'un nombre écrit en base 10.* »

```
int ndigits (int n) {
  if (n < 0) n = -n ;
  return (1 + (int) log10 (n)) ;
}
```

- Dans le code : $\log_{10}$.
- Définie uniquement sur $\mathbb{R}^{*}_{+}$.
- Idée de test : 0.

```
$ ./a.out 0
-2147483647
```

Tests structurels (3/3)

« Calculer le nombre de chiffres d'un nombre écrit en base 10. »

```
int ndigits (int n) {
  if (n == 0) return 1 ;
  if (n < 0) n = -n ;
  return (1 + (int) log10 (n)) ;
}
```

```
$ ./a.out -100
3
$ ./a.out 0
1
$ ./a.out 90
2
```