CS54 : Algorithmique

I/ Fondations théoriques & pratiques de la programmation

A/ Introduction

1/ Problèmes

→ Enoncé générique, i.e. "déterminez la valeur minimale d'un ensemble d'entiers", qui doit être clair, précis, sans ambiguïtés et complet

→ Sont instanciés en leur donnant des données, i.e. "déterminez la valeur minimale de $\{17, 6, 3, 51\}$"

2/ Programmes

→ Ensemble d'actions qui vont s'exécuter dans un ordre donné.

→ Réalisable par un ordinateur

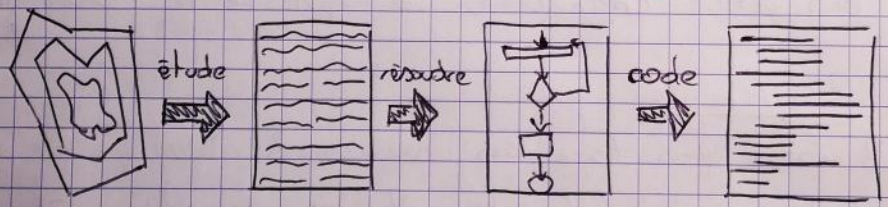
3/ Algorithmes

→ Description précise du processus de résolution du problème

→ Doit être compréhensible (par l'humain)

→ Ne dépend pas du langage de prog, du compilateur ou de la machine

→ Peut être représenté en tant que diagramme



Problème Spécification Algorithme Programme

4/ Computer Science VS Software Engineering

→ Computer Science is the science of abstraction - creating the right model for a problem and devising the appropriate mechanizable technique to solve it

-Aho and Ullman

→ Centres d'intérêt en CS :

- calculabilité (pour un programme donné, peut-on démontrer qu'il

existe un algorithme capable de le résoudre ? Quels sont les problèmes pour lesquels il n'y a pas d'algorithme ?)

- Complexité :
  → Combien de temps l'algorithme va-t-il prendre pour trouver ?
  → Quelle sera sa consommation de mémoire ?
  → Mon algorithme est-il optimal ?
- Exactitude :
  → Mon algorithme va-t-il toujours trouver une solution ?
  → Pouvons-nous être certains que la solution est la bonne ?

→ En ingénierie logicielle on produit des réponses techniques au problème donné

B/ Créer des algorithmes pour les problèmes complexes

- Critères de choix parmis les algorithmes :
  - Exactitude
  - Simplicité : KISS, faire de la décomposition ou de l'abstraction (travailler sans les détails)
  - Efficacité : rapide & utilise peu de ressources
  - Stabilité : des petits changements d'input ne le cassent pas

C/ Mesure de l'efficacité

→ On pourrait le coder puis faire des mesures & statistiques, mais beaucoup de facteurs externes (machine, langage, compilateur, OS, ...) donc pas assez générique

→ On va donc faire une estimation mathématique, qui va compter la quantité d'instructions basiques en fonction de la taille de l'input.

1/ Mesures

- t_{min} : nombre d'instruction pour l'entrée la plus simple
- t_{max} : nombre d'instructions pour l'entrée la plus complexe

• t_{avg} : nombre d'instructions moyen, pondéré par la probabilité

2/ Complexité asymptotique

→ On ignore les éléments les moins importants pour avoir une complexité claire, i.e. n, n^3, 2^n, 1, ...

→ $O(n)$ est la limite supérieure de la complexité, à partir d'un n_0 donné, à une constante multiplicative près

→ $\Omega(n)$ est pareil que $O(n)$ mais en limite inférieure

→ $\Theta(n)$ est la meilleure estimation de la complexité. On a $\Omega(n) \leq \Theta(n) \leq O(n)$, $\forall n \geq n_0$

→ Règles de calcul :
- la complexité d'une suite d'instructions est la somme de chacunes d'elles
- la complexité des instructions basiques est $\Theta(1)$
- si on a des conditions, prendre le pire cas
- si on a des boucles, c'est la complexité du corps × le nombre d'itérations

II/ Récursion

→ fonction dont son corps contient au moins un appel à elle-même (cas général)

→ toujours un cas de base où elle ne s'appelle pas, sinon ça fait une boucle infinie

• Résoudre un problème récursivement :
1. Déterminer le paramètre sur lequel la récursion opère : type de base (i.e. int) ou type récursif (i.e. arbre binaire) ?
2. Trouver les cas de base
3. Trouver comment réduire le périmètre de recherche grâce à un appel récursif (prendre des exemples si on y arrive pas) et en

déduire le cas de base
⑤ Vérifier que les cas de base sont toujours atteints

III/ Test

A/ Terminologie

→ Éviter le mot "bug" car imprécis
→ Erreur : comportement incorrect du logiciel
  → Erreur transitoire : seulement avec certains input
  → Erreur permanente : pour tout input
→ Faute : cause de l'erreur
→ Echec : instance particulière d'une erreur générale, causée par une faute

B/ Techniques de contrôle qualité

- évitement : trouver les fautes avant la relecture
- détection : trouver les fautes sans se remettre des erreurs
- résistance aux erreurs : quand le système peut se remettre d'un échec tout seul

C/ Terminologie (II)

- Composant : partie d'un système qui peut être isolé pour le tester (par le biais d'un stub et d'un driver)
- Test Case : ensemble {inputs, résultats attendus} à tester. Les tests sont booléens : est-ce que en donnant input ça renvoit résultat attendu ? (peut inclure des exceptions & codes d'erreur)
- Test Stub : programmes simulants des composants dont ce qu'on veut tester dépendent. I.E. : on dev côté client et pour éviter d'attendre les résultats d'appels serveurs on hardcode temporairement les fonctions d'appel serveur
- Test Driver : programme simulant des composants qui ont besoin du composant testé

CS54 : Algorithmique

- Tester : fait d'exécuter un programme dans le but d'en trouver ses défauts
- Stratégies de test : plans pour dire quand effectuer quels tests

D/ Techniques de test

1/ White box testing

→ utiliser nos connaissances des composants pour créer les inputs de test, i.e. la structure est un array de taille 256, tester
→ maximiser la couverture en minimisant le nombre de tests
→ se concentrer sur les états internes des composants
→ critère de couverture : tout le cheminement est correct :
- toutes les déclarations
- toutes les conditions, y compris les composées
- tout un flux de données y fonctionne
- toutes les boucles, en cas d'itération 0x, 1x ou Nx

2/ Black box testing

→ les composants sont des boîtes noires, on les teste comme si on ne connaissait pas leur fonctionnement
→ on regarde si en donnant des inputs on a bien les output attendus :
- toutes les exceptions
- toutes les données qui génèrent un résultat différent
- toutes les valeurs limites

E/ Stratégies de test

- Tests unitaires : ensemble de tests pour 1 composant
- Tests d'intégration : tests entre plusieurs composants, pour

s'assurer qu'ils sont bien compatibles
- Tests de régression : s'assurer que ce qui marchait avant fonctionne toujours après une MAJ, en rejouant les tests
- Tests d'acceptation : faire essayer le produit aux utilisateurs finaux pour s'assurer que ce qu'on a fait est bien ce qui était demandé et que ça fonctionne
  → Alpha : fait en présence des développeurs, dans un environnement contrôlé
  → Beta : dans le monde réel, sur les machines utilisateurs sans les devs
- Tests de récupération : forcer le système à faire une erreur et s'assurer qu'il s'en remette bien
- Stress Test : tester en conditions extrêmes, i.e. surcharge serveur
- Test de performances en activité : temps, mémoire utilisée.
  ⚠ Dernière chose à faire ! "D'abord faites-le, puis faites le bien, puis faites le vite"
- Back-to-back testing : comparer les résultats entre les versions
  → commande bisect avec git

IV. / Dérécursivation

→ La récursivité est très souvent moins performante que les solutions itératives
→ Toute fonction récursive peut être dérécursivée

A/ Sur un algo en tail recursion

Tail recursion : où le résultat retourné par le cas de base n'est pas modifié. i.e :
- return F(x-1) : OK
- return F(x-1)/2 : non
- return F(x-1) + F(x-2) : non

→ dérécursivation

```
def f(x):
    if cond(x):
        return x+3
    else
        x = transformer(x)
        return f(x)
```

→

```
def f(x):
    while not cond(x):
        x = transformer(x)
    return x+3
```

B/ Sur un algo pas en tail recursion

1/ Si c'est associatif et commutatif

→ essayer de modifier l'opération qui se fait en remontée pour qu'elle se fasse en descente ! (donc on transforme en tail)

→ se fait souvent en ajoutant un paramètre supplémentaire (donc faire une fonction -helper)

→ ce paramètre supplémentaire est appelé accumulateur. Il peut y en avoir plusieurs : autant que d'opérations

→ exemple :

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

```
def fact(n): ← ne contente d'appeler le helper initialisé
    return fact_helper(n, 1)

def fact_helper(n, acc):
    if n==0:
        return acc   ← les cas de base retournent un résultat issu des accumulateurs
    else:            plus d'opérations en remontée !
        return fact_helper(n-1, acc*n)
```

→ quand c'est en tail récursion, on sait faire !

2/ Quand la première solution ne fonctionne pas ...

→ on va simuler la stack

→ concrètement :

```
def f(x):
    if cond(x):
        return x+3
    else:
        x = transforme(x)
        return modifier(x, f(x))
```

→ si transformer(x) est réversible, pas besoin de stack !

```
def f(x):
    s = deque()  ← stack de Python
    while not cond(x):
        s.append(x)
        x = transformer(x)
    r = x+3  ← cas de base
    while s:  ← tant que la stack n'est pas vide
        x = s.pop()
        x = transformer(x)
        r = modifier(x, r)
    return r
```

V/ Backtracking

- Recherche combinatoire : recherche dans des ensembles finis d'une solution respectant toutes les contraintes
- Optimisation combinatoire : recherche dans des ensembles finis la solution étant le meilleur choix respectant toutes les contraintes

→ Plusieurs approches :

- recherche exaustive, en testant toutes les possibilités
  → extrêmement long
- backtracking : voir le cheminement vers la solution comme un arbre où on essaye chaque branche, mais dès qu'on se rend compte qu'elle est impossible on la coupe et on change de branche (on revient au noeud de branche précédent)

→ quelques principes du backtracking :

- l'ordre de choix d'exploration des branches peut up les perfs
- la complexité peut vite exploser quand même