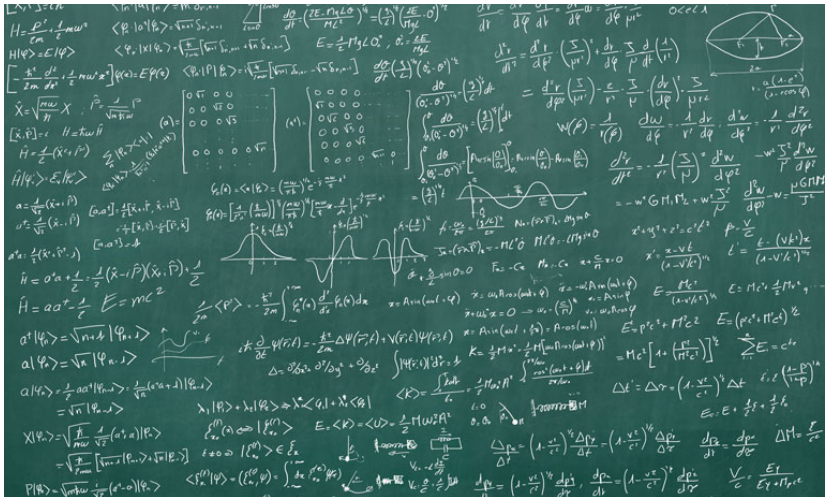
Informatique 2M

A. Ridard

Algorithmique 2



Gymnase Yverdon

canton de vaud

Table des matières

I. Recherche dichotomique

1. Recherche linéaire dans un tableau

Voici un algorithme de recherche d'un élément dans un tableau (une liste Python).

Algorithme 1 : recherche linéaire

Entrées : un tableau t, un élément v
Sorties : l'indice de l'élément dans le tableau s'il est présent, et None sinon

```
n ← longueur(t) ;
pour i de 0 à n-1 (inclus) faire
    si t[ i ] = v alors
        res ← i ;
    fin
fin
res ← None ;
```

Quelle est sa « complexité[a] » au regard du nombre de comparaisons effectuées ?

a. On utilise le nombre d'opérations élémentaires effectuées pour mesurer l'efficacité d'un algorithme, plutôt que sa vitesse d'exécution qui dépend des capacités de calcul de la machine.

On peut améliorer cet algorithme en remplaçant la boucle **for** par une boucle **while**.

Algorithme 2 : recherche linéaire optimisée

Entrées : un tableau t, un élément v
Sorties : l'indice de l'élément dans le tableau s'il est présent, et None sinon

```
n ← longueur(t) ;
i ← 0 ;
tant que i < n et t[ i ] != v faire
    i ← i + 1 ;
fin
si i < n alors
    res ← i ;
sinon
    res ← None ;
fin
```

A-t-on vraiment amélioré la « complexité » de cet algorithme ?
On peut s'intéresser au meilleur/pire des cas et au comportement « moyen ».

Appliquer cet algorithme avec $t = [1, 1, 0, 3]$ et

- $v = 1$
- $v = 0$
- $v = 7$

2. Recherche dichotomique dans un tableau *trié*

Si le tableau est trié [1], alors la recherche peut être nettement plus efficace !

Algorithme 3 : recherche dichotomique

Entrées : un tableau trié t, un élément v
Sorties : l'indice de l'élément dans le tableau s'il est présent, et None sinon

```
n ← longueur(t) ;
g ← 0 ;
d ← n-1 ;
res ← None ;
tant que g <= d faire
    m ← (g + d) // 2 ;
    si t[ m ] < v alors
        // on peut limiter la recherche à t[ m+1..d ]
        g ← m + 1 ;
    fin
    si t[ m ] > v alors
        // on peut limiter la recherche à t[ g..m-1 ]
        d ← m - 1 ;
    fin
    si t[ m ] = v alors
        // on a trouvé l'élément puisque l'on a dans ce cas t[ m ] = v
        res ← m ;
        break ;
    fin
fin
```

Appliquer cet algorithme avec $t = [0, 1, 1, 2, 3, 5, 8, 13, 17, 21]$ et

- $v = 1$
- $v = 13$
- $v = 7$

Voici le nombre maximal [2] d'itérations k selon la taille n du tableau.

n	k
10	4
100	7
1000	10
1 000 000	20
1 000 000 000	30

La complexité de cet algorithme est égale au plus petit entier k vérifiant $2^k > n$.
Son **ordre de grandeur** est $\log_2(n)$ ou encore [a] $ln(n)$ (largement inférieur à n).

On dit que la recherche dichotomique a une **complexité logarithmique**, qui est bien meilleure que la **complexité linéaire** du premier algorithme.

a. Vous verrez cette année en maths que $\log_2(n) = \dfrac{\ln(n)}{\ln(2)}$

[1]. On peut utiliser pour cela un des tris vus en première année.
[2]. correspondant au pire des cas

3. « Diviser pour régner » et récursivité

Le principe « diviser pour régner » consiste à décomposer un problème en sous-problèmes plus petits, puis à les résoudre **en appliquant le même principe** autant de fois que nécessaire, et enfin à combiner les résultats des sous-problèmes pour en déduire le résultat du problème initial. Ce principe invite à penser la solution de manière **récursive**.

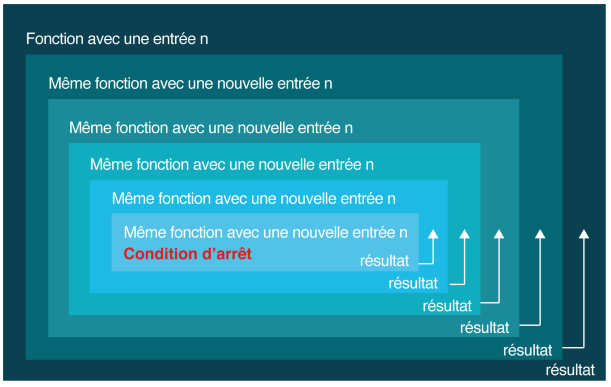
Imaginons que nous disposions d'une fonction ***recherche(t, v, g, d)*** qui renvoie l'indice de l'élément ***v*** dans le tableau ***t*** (trié), s'il est présent dans ***t[g..d]***, et None sinon. Le résultat du problème initial serait alors fourni par ***recherche(t, v, 0, n-1)*** où n désigne la longueur de t.

Comment peut-on définir cette fonction ***recherche(t, v, g, d)***?

En l'appelant elle-même...

Fonction récursive

Une fonction qui s'appelle elle-même est appelée fonction récursive. Il s'agit d'une mise en abîme, d'une définition circulaire. Lorsqu'on entre dans la fonction, des opérations sont exécutées et on fait à nouveau appel à la fonction, mais cette fois-ci avec une nouvelle entrée qui nous « rapproche » du résultat :



N'oubliez pas la condition d'arrêt

Les deux ingrédients indispensables à toute fonction récursive sont :

- un appel à la fonction elle-même
- une condition d'arrêt qui permet de terminer les appels imbriqués

Algorithme 4 : recherche (récursive)

Entrées : t, v, g, d
Sorties : l'indice de v dans t (trié), s'il est présent dans t[g..d], et None sinon

```
// 1ère condition d'arrêt
si g > d alors
    res ← None ;
fin
m ← (g + d) // 2 ;
si t[ m ] < v alors
    recherche(t, v, m+1, d) ;
fin
si t[ m ] > v alors
    recherche(t, v, g, m-1) ;
sinon
    // 2ème condition d'arrêt lorsque t[ m ] = v
    res ← m ;
fin
```

Donner la séquence des appels à la fonction **recherche** pour résoudre le problème avec t = [0, 1, 1, 2, 3, 5, 8, 13, 17, 21] et

- $v = 1$
- $v = 13$
- $v = 7$

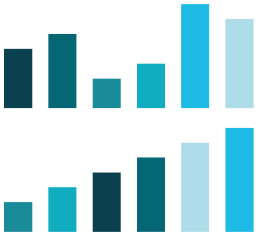
La recherche dichotomique est un exemple simple du principe « diviser pour régner » car il ne nécessite pas de combiner les résultats des sous-problèmes pour obtenir le résultat du problème initial.

Nous allons maintenant nous intéresser à un autre exemple de ce principe, plus délicat à cause de la combinaison qui sera cette fois-ci nécessaire.

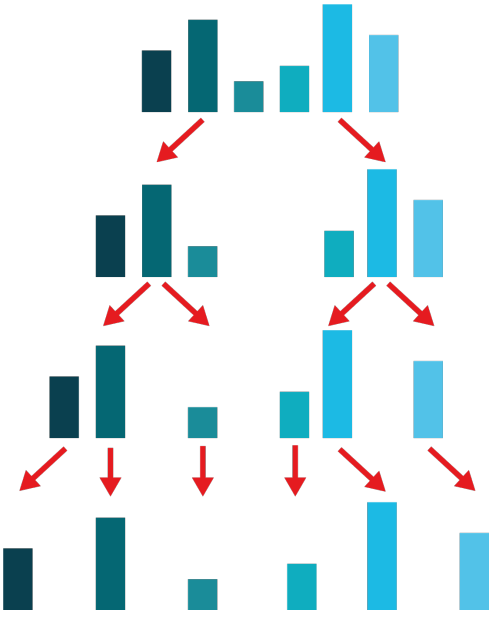
II. Tri par fusion

1. Principe en images

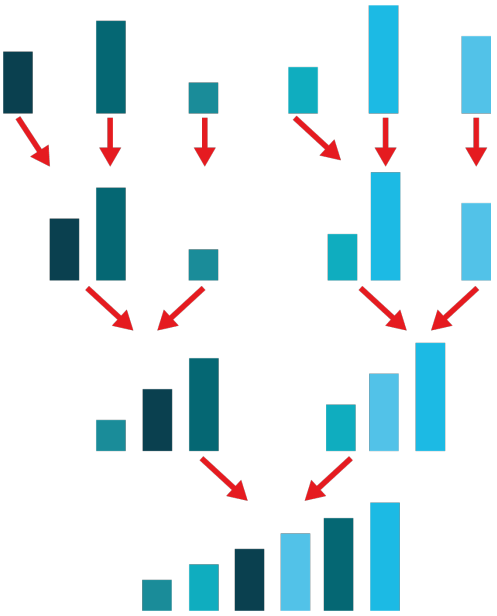
Comme en première année, on souhaite trier une liste (dans l'ordre croissant).



D'abord, on divise en sous-problèmes plus petits...



Puis, on combine (fusionne) pour obtenir le résultat du problème initial...



Détailler les différentes étapes du tri par fusion sur la liste [3, 4, 1, 7, 2].

2. Écriture de l'algorithme

Algorithme 5 : triFusion

Entrées : lst non vide
Sorties : lst triée

```
// condition d'arrêt
si longueur(lst) = 1 alors
    res ← lst ;
    break ;
fin
lst1, lst2 ← coupe(lst) ;
res ← fusion(triFusion(lst1), triFusion(lst2)) ;
```

Les deux fonctions suivantes restent évidemment à définir :

- **coupe(lst)** sépare une liste en deux listes de même longueur à un près
- **fusion(lst1Tr, lst2Tr)** combine deux listes triées pour en renvoyer une triée

Algorithme 6 : coupe

Entrées : lst de longueur supérieure ou égale à 2
Sorties : deux sous-listes de lst de même longueur à un près

```
n ← longueur(lst) ;
m ← n // 2 ;
res ← lst[ 0..m ], lst[ m+1..n-1 ] ;
```

Algorithme 7 : fusion

Entrées : deux listes triées lst1Tr, lst2Tr
Sorties : liste triée par fusion de lst1Tr et lst2Tr

```
n1 ← longueur(lst1Tr) ;
n2 ← longueur(lst2Tr) ;
// 1ère condition d'arrêt
si n1 = 0 alors
    res ← lst2Tr ;
    break ;
fin
// 2ème condition d'arrêt
si n2 = 0 alors
    res ← lst1Tr ;
    break ;
sinon
    // on appelle la fonction elle-même
    si lst1Tr[ 0 ] <= lst2Tr[ 0 ] alors
        res ← [ lst1Tr[ 0 ], fusion(lst1Tr[ 1..n1-1 ], lst2Tr ] ;
    sinon
        res ← [ lst2Tr[ 0 ], fusion(lst1Tr, lst2Tr[ 1..n2-1 ] ] ;
    fin
fin
```

Limite de la récursivité

Si on cherche à trier une liste de plus de 1000 éléments avec notre fonction[a] **triFusion**, on obtient l'erreur **RecursionError** qui signifie que l'on a fait un trop grand nombre d'appels récursifs imbriqués.

Plus précisément, c'est la fonction **fusion** qui est responsable de cette erreur. Une solution consisterait à augmenter le nombre maximal d'appels, fixé à 1000 par défaut, avec **setrecursionlimit**.

La fonction **triFusion**, bien que récursive, ne conduira en revanche jamais à l'erreur **RecursionError**. En effet, la taille de la liste étant divisée par deux à chaque fois, il faudrait une liste de plus de 2^{1000} éléments pour conduire à une erreur. La mémoire de notre machine ne nous permet pas de construire une liste aussi grande !

a. En traduisant en Python les algorithmes écrits en pseudo-code, on obtient les fonctions naturellement associées.

Traduire en Python les algorithmes **triFusion** et **coupe** pour définir les fonctions naturellement associées.

3. Efficacité

Voici un tableau comparant les performances du tri par sélection et du tri par fusion.

taille n	sélection	fusion
1000	0,06 s	0,01 s
2000	0,13 s	0,03 s
4000	0,44 s	0,05 s
8000	1,78 s	0,11 s
16000	6,79 s	0,29 s

Son **ordre de grandeur** est $n\ln(n)$ (largement inférieur à n^2). Cette **complexité linéarithmique** est bien meilleure que la **complexité quadratique** des tris élémentaires, elle est même optimale pour le tri.

III. Algorithmes gloutons

1. Problème du voyageur

Supposons que l'on ait déterminé une liste de villes à visiter et que l'on recherche un itinéraire minimisant la distance totale parcourue. On s'autorisera à visiter les villes dans n'importe quel ordre, mais aucune ne doit être négligée, et il faudra à la fin revenir à la ville de départ.

Par exemple, nous partons de Nancy et devons nous rendre à Metz, Paris, Reims et Troyes avant de revenir à Nancy. Le tableau des distances routières kilométriques entre ces différentes villes est le suivant.

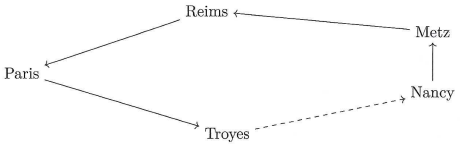
	Nancy	Metz	Paris	Reims	Troyes
Nancy		55	303	188	183
Metz	55		306	176	203
Paris	303	306		142	153
Reims	188	176	142		123
Troyes	183	203	153	123	

Une manière simple d'aborder le problème consiste à énumérer tous les ordres possibles, à calculer pour chacun la distance correspondante, pour sélectionner ensuite la plus petite. On a ici 24 itinéraires possibles dont 12 sont détaillés dans le tableau suivant où le départ et l'arrivée à Nancy sont sous-entendus.

Circuit	Détail étapes	Total
Metz - Paris - Reims - Troyes	$55+306+142+123+183$	**809**
Metz - Paris - Troyes - Reims	$55+306+153+123+188$	**825**
Metz - Troyes - Paris - Reims	$55+203+153+142+188$	**741**
Troyes - Metz - Paris - Reims	$183+203+306+142+188$	**1022**
Troyes - Metz - Reims - Paris	$183+203+176+142+303$	**1007**
Metz - Troyes - Reims - Paris	$55+203+123+142+303$	**826**
Metz - Reims - Troyes - Paris	$55+176+123+153+303$	**810**
Metz - Reims - Paris - Troyes	$55+176+142+153+183$	**709**
Reims - Metz - Paris - Troyes	$188+176+306+153+183$	**1006**
Reims - Metz - Troyes - Paris	$188+176+203+153+303$	**1023**
Reims - Troyes - Metz - Paris	$188+123+203+306+303$	**1123**
Troyes - Reims - Metz - Paris	$183+123+176+306+303$	**1091**

Les 12 autres itinéraires correspondent chacun à l'un des 12 du tableau emprunté dans le sens inverse, ce qui ne change en rien la distance.

Une inspection de cette table indique qu'un circuit passant par toutes les villes voulues peut être bouclé en 709 km.



Cette méthode dite **exhaustive** permet de trouver la solution optimale, mais elle ne « passe pas à l'échelle ». En se fixant une liste plus longue de villes à visiter, le nombre de circuits différents à analyser devient vite beaucoup trop important, et ce même pour les capacités de calcul d'un ordinateur. Nous avons ainsi 12 circuits pour 4 villes (hors ville de départ), presque 2 millions pour 10 villes, plus de 3 milliards pour 13 villes, et plus d'un milliard de milliards pour 20 villes (la valeur exacte est la moitié de la factorielle du nombre de villes). Pire, ce phénomène n'est pas dû à la naïveté de notre algorithme, mais plutôt à la « difficulté » du problème [3].

Les différents algorithmes s'appliquant à un tableau de longueur n peuvent impliquer des nombres d'opérations très différents, proportionnels à :

- $\ln(n)$: recherche dichotomique
- $n\ln(n)$: tri par fusion
- n : parcours d'un tableau
- n^2 : tri par sélection
- ...

[3]. Il appartient à la classe des problèmes dits *NP-complets* caractérisés par deux conditions :
- on peut vérifier la validité d'une solution en un temps *polynomial* (raisonnable)
- l'exploration des solutions possibles nécessite un temps *exponentiel* (impossible en pratique)

Pour mesurer l'importance de cette différence, voici un tableau des tailles n donnant un nombre d'opérations proche du milliard, c'est à dire un nombre d'opérations qu'il est envisageable de faire exécuter à un programme.

nombre d'opérations	n maximal
$\ln(n)$	$10^{300\ 000\ 000}$
$\sqrt{n}$	10^{18}
n	10^{9}
n^2	31600
n^3	1000
2^n	30
$n!$	12

Face à de tels problèmes d'optimisation, impossibles à explorer de manière exhaustive, on se tourne en général vers les **heuristiques**. Ce sont des méthodes qui fournissent « rapidement » une solution approchée, c'est à dire une solution « acceptable », mais pas nécessairement optimale.

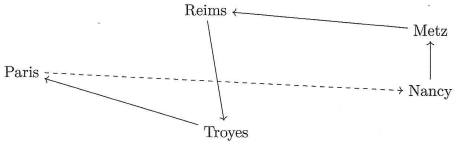
Un **algorithme glouton** est une heuristique qui construit une solution globale par une succession de choix en prenant systématiquement l'option donnant la « meilleure » solution locale.

C'est un peu comme si on cherchait à atteindre le plus haut sommet d'une montagne, entouré de brouillard, et que l'on prenait nos décisions successives sur le chemin à emprunter uniquement en fonction de ce que l'on peut voir juste autour de nous. En choisissant par exemple systématiquement le chemin le plus pentu, on peut espérer s'approcher du plus haut sommet. Mais une fois arrivé en haut d'un sommet, on ne peut savoir si c'est bien le plus haut...

Heureusement, il est possible pour certains problèmes de calculer l'ordre de grandeur du rapport entre la solution optimale et la solution gloutonne dans le pire des cas, de manière à avoir quelques garanties !

Appliquons l'approche gloutonne suivante à notre problème du voyageur. Partant de la ville de départ, aller à la ville la plus proche, puis à la ville la plus proche de cette dernière parmi les villes non encore visitées, et ainsi de suite.

Partant de Nancy, nous irons donc en premier lieu à Metz, distante de 55 km. Ensuite à Reims en 176 km, puis à Troyes en 123 km, et enfin à Paris en 153 km, avec un ultime retour à Nancy en 303 km. On complète ainsi notre circuit en 810 km.



On obtient alors rapidement une solution non optimale, mais tout à fait acceptable.

Écrire en pseudo-code l'algorithme glouton **voyageur** prenant en entrée :

- la liste des n villes à visiter, repérées par leur indice de 0 à $n-1$
- le tableau $dist$ à 2 dimensions[a] tel que $dist[i][j]$ donne la distance de la ville d'indice i à la ville d'indice j
- l'indice de la ville de départ

Il délègue le choix de la ville suivante à l'algo **plusProche** prenant en entrée :

- l'indice de la ville actuelle
- le tableau $dist$
- la liste des booléens avec *True* aux indices des villes déjà visitées

a. Une liste de listes en Python

L'algo **plusProche** est constituée d'une boucle examinant les villes tour à tour en écartant les villes déjà visitées, et en mémorisant dans une variable *pp* l'indice de la ville non visitée la plus proche vue jusque là. Cette variable est initialisée avec *None* puisque l'on ne connaît pas a priori l'indice de la première ville éligible.

Algorithme 8 : plusProche

Entrées :
Sorties :

L'algo **voyageur** utilise une liste *villesVis* de booléens (celle en entrée de **plusProche**) et une variable *villeAct* mémorisant l'indice de la ville actuelle. Il est constitué d'une boucle qui $n-1$ fois marque comme visitée la ville actuelle puis sélectionne la suivante.

Algorithme 9 : voyageur

Entrées :
Sorties :

2. Problème du sac à dos

Arsène L., équipé d'un sac à dos, a devant lui un ensemble d'objets de valeurs et de poids variés. Il doit choisir les objets qu'il va emporter de manière à maximiser la valeur totale, sans dépasser 10 kg. Voici donc un nouveau problème d'optimisation!

Les 4 situations que nous allons étudier sont résumées dans les tableaux suivants :

obj	kg	chf
A	8	4800
B	5	4000
C	4	3000
D	1	500

obj	kg	chf
A	6	4800
B	5	3500
C	4	3000
D	1	500

obj	kg	chf
A	9	8100
B	6	7200
C	5	5500
D	4	4000
E	1	800

obj	kg	chf
A	7	9100
B	6	7200
C	4	4800
D	3	2700
E	2	2600
F	1	200

Approche exhaustive

Pour chacune des 4 situations, indiquer toutes les solutions possibles (objets pris et valeur totale), puis déterminer la solution optimale.

Algorithmes gloutons

Arsène L. ne voulant pas arriver en retard à son rendez-vous avec la comtesse C., il doit choisir rapidement les objets à emporter. Pour chacune des 4 situations, appliquer les 3 algorithmes gloutons suivants :

1. Choisir les objets par ordre de valeur décroissante parmi ceux qui ne dépassent pas la capacité restante.
2. Choisir les objets par ordre de poids croissant.
3. Choisir les objets par ordre de rapport valeur/poids décroissant parmi ceux qui ne dépassent pas la capacité restante.

Quelle stratégie gloutonne est à privilégier?

Piéger les stratégies gloutonnes

Que se passe-t-il dans les cas suivants ?

1. **Algo glouton 1** avec :
   - un objet de 10 kg et de valeur 10
   - dix objets de 1 kg et de valeur 9
2. **Algo glouton 2** avec :
   - dix objets de 1 kg et de valeur 1
   - un objet de 10 kg et de valeur 100
3. **Algo glouton 3** avec :
   - un objet de 6 kg et de valeur 60 (rapport 10)
   - deux objets de 5 kg et de valeur 45 (rapport 9)

3. Problème de planification

Supposons avoir une liste d'activités, chacune associée à un créneau horaire défini par une heure de début et une heure de fin. Deux activités sont compatibles si leurs créneaux ne se recouvrent pas. On souhaite sélectionner un nombre maximal d'activités, évidemment compatibles entre elles. Encore un problème d'optimisation !

Considérons par exemple la situation suivante :

	activité 1	activité 2	activité 3	activité 4	activité 5
heure de début	8	12	9	14	11
heure de fin	13	17	11	16	12

Approche exhaustive

Indiquer toutes les solutions possibles (activités et nombre), puis déterminer la(les) solution(s) optimale(s).

Algorithme glouton

On propose un algorithme glouton pour sélectionner les activités en commençant par le début de la journée : choisir l'activité dont l'heure de fin arrive le plus tôt (parmi les activités dont l'heure de début est bien postérieure aux créneaux des activités déjà choisies).

Appliquer cette stratégie[a] à notre situation.

a. On peut démontrer que cet algorithme glouton fournit toujours une solution optimale !