TD 01 – Introduction à l'algorithmique (corrigé)

(GrandSaut) **Exercice 1.** *Grand Saut*

Le problème est de déterminer à partir de quel étage d'un immeuble, sauter par une fenêtre est fatal. Vous êtes dans un immeuble à n étages (numérotés de 1 à n) et vous disposez de k étudiants. Les étudiants sont classés par notes de partiel croissantes. Il n'y a qu'une opération possible pour tester si la hauteur d'un étage est fatale : faire sauter le premier étudiant de la liste par la fenêtre. S'il survit, vous pouvez le réutiliser ensuite (évidemment, l'étudiant survivant reprend sa place initiale dans la liste triée), sinon vous ne pouvez plus.
Vous devez proposer un algorithme pour trouver la hauteur à partir de laquelle un saut est fatal (l'algorithme doit renvoyer $(n+1)$ si on survit encore en sautant du n-ème étage) en faisant le minimum de sauts.

1. Si $k \geq \lceil \log_2(n) \rceil$, proposer un algorithme en $\mathcal{O}(\log_2(n))$ sauts.

   ☞ La complexité en $O(\log_2(n))$ qui est indiquée nous aiguille vers une dichotomie. En effet, en supposant que l'on a $k \geq \lceil \log_2(n) \rceil$, on obtient le résultat sur les étages de i à j en jetant un étudiant depuis le $n^{\text{ème}}$ étage, où $n = j - i/2$, puis en itérant le procédé sur les étages de i à $n-1$ si la chute à été fatale, et sur les étages de n à j dans le cas contraire. La méthode dichotomique nous garantit alors que l'on obtient le bon résultat (lorsqu'il ne reste plus qu'un seul étage, c'est-à-dire lorsque l'on calcule pour les étages de $i = j$), et ce avec une complexité logarithmique dans le pire des cas.

2. Si $k < \lceil \log_2(n) \rceil$, proposer un algorithme en $\mathcal{O}(k + \frac{n}{2^{k-1}})$ sauts.

   ☞ Puisqu'ici on ne dispose que de $k < \lceil \log_2(n) \rceil$ étudiants, on ne peut pas appliquer directement la méthode dichotomique proposée précédemment. Cependant, on va remédier au problème de manière simple en appliquant une recherche dichotomique avec $k-1$ étudiants, de manière à délimiter un intervalle d'étages dans lequel se trouve l'étage recherché. On se sert alors du dernier étudiant restant pour parcourir l'intervalle de façon linéaire, donc en le jetant de chaque étage en partant du plus bas de l'intervalle, jusqu'au plus haut. Après avoir jeté les $k-1$ premiers étudiants, si l'on n'a pas encore trouvé le bon étage, il reste exactement $n/2^{k-1}$ étages dans l'intervalle de recherche, d'où une complexité dans le pire des cas en $O(k + n/2^{k-1})$ sauts.

3. Si $k = 2$, proposer un algorithme en $\mathcal{O}(\sqrt{n})$ sauts.

   ☞ Dans le cas particulier où l'on a $k = 2$, on ne veut pas avoir à tester chaque étage de façon linéaire, c'est pourquoi on va reprendre à notre compte les idées précédentes, et notamment celle qui consiste à délimiter un intervalle de recherche. Nous découpons donc l'ensemble des étages en "tranches" de $\sqrt{n}$ étages, avant de jeter le premier étudiant de chacun des étages de début de tranche. Lorsque l'étudiant y laisse sa peau, on se ramène au dernier étage n testé qui ne soit pas fatal, et on n'a plus qu'à parcourir de manière linéaire l'intervalle allant de l'étage $n+1$ à l'étage fatal trouvé précédemment. On a ainsi deux séries d'essais en $O(\sqrt{n})$, et donc une complexité finale dans le pire des cas également en $O(\sqrt{n})$ sauts.

(Bricolage) **Exercice 2.** *Bricolage*

Dans une boîte à outils, vous disposez de n écrous de diamètres tous différents et des n boulons correspondants. Mais tout est mélangé et vous voulez appareiller chaque écrou avec le boulon qui lui correspond. Les différences de diamètre entre les écrous sont tellement minimes qu'il n'est pas possible de déterminer à l'œil nu si un écrou est plus grand qu'un autre. Il en va de même avec les boulons. Par conséquent, le seul type d'opération autorisé consiste à essayer un écrou avec un boulon, ce qui peut amener trois réponses possibles : soit l'écrou est strictement plus large que le boulon, soit il est strictement moins large, soit ils ont exactement le même diamètre.

1. Écrire un algorithme simple en $\mathcal{O}(n^2)$ essais qui appareille chaque écrou avec son boulon.

   ☞ Pour appareiller les boulons et les écrous, il suffit de prendre un boulon arbitrairement, de le tester avec tous les écrous.On trouve alors le bon en au plus n tests et il suffit alors de recommencer avec tous les autres boulons.On obtient donc le résultat en au plus $\frac{n(n-1)}{2} = O(n^2)$ tests.

2. Supposons qu'au lieu de vouloir appareiller tous les boulons et écrous, vous voulez juste trouver le plus petit écrou et le boulon correspondant. Montrer que vous pouvez résoudre ce problème en moins de $2n-2$ essais.

☞ Le principe est de numéroter les boulons et les écrous de 1 à n, de manière arbitraire, et de faire progresser des compteurs (par exemple i et j) pour marquer le minimun courant dans l'une des catégories, et la structure en cours de test dans l'autre.

---

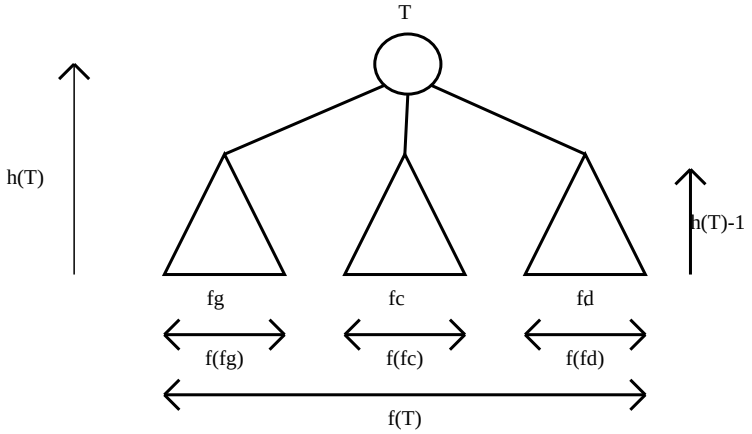
début
  tant que $(i \leq n)$ && $(j \leq n)$ **faire**
    si i=j=n **alors** sortir de la boucle ;
    si $ecrou.i = boulon.j$ **alors** s'en souvenir et faire $i := i+1$;
    si $ecrou.i < boulon.j$ **alors** $j := j+1$; $min = ecrou$;
    si $ecrou.i > boulon.j$ **alors** $i := i+1$; $min = boulon$;

A la fin de cette boucle,
- si min=écrou, l'écrou i est le plus petit.
- si min=boulon, le boulon j est le plus petit. Et dans les deux cas, on sait déjà quel est l'élément qui correspond : en effet, le compteur de l'autre catégorie vaut n+1 (ou n dans un cas spécial),et on a donc déjà rencontré le minimum. la seule raison possible pour laquelle il n'a pas été conservé est donc qu'il ait été testé avec l'autre minimum. Pour le cas spécial ou i=j=n, le dernier test est inutile : en effet, on sait que l'un des deux, est minimum, et que une fois le boulon minimum atteind, j reste fixe : d'où le boulon n est minimum, et son homologue a soit déjà été trouvé, soit est l'écrou restant.

Cette boucle effectue donc au plus $2 * n - 2$ tests.

3. Prouver que tout algorithme qui appareille tous les écrous avec tous les boulons doit effectuer $\Omega(n \log n)$ essais dans le pire des cas.

☞ On note $f(T)$ le nombre de feuilles de l'arbre et $h(T)$ sa hauteur. Si T est un arbre ternaire $h(T) \geq \lceil log_3 f(T) \rceil$. Par induction sur la hauteur de T :
- Pour $h(T) = 0$ l'arbre à une feuille donc on a bien $h(T) \geq \lceil log_3 f(T) \rceil$.
- Pour $h(T) > 0$ un arbre ternaire a trois fils, le fils gauche fg, le fils central fc, et le fils droit fd de hauteur inférieure à $h(T) - 1$. On suppose que $h(T) \geq \lceil log_3 f(T) \rceil$ est vrai pour $h(T) \leq k$ k étant fixé on veut démontrer que cette propriété est vrai aussi pour $h(T) = k + 1$. Les feuilles d'un arbre sont aussi celles de ses fils. Donc



$$f(T) = f(fg) + f(fc) + f(fd)$$

de plus :

$$h(T) \leq h(fd) + 1$$
$$h(T) \leq h(fg) + 1$$
$$h(T) \leq h(fc) + 1$$

or par induction comme $h(T) = k + 1$, $h(fg) \leq k$, $h(fc) \leq k$,et $h(fd) \leq k$ on a :

$$k = h(fc) \leq \lceil log_3 f(fc) \rceil$$
$$h(fd) \leq \lceil log_3 f(fd) \rceil$$
$$h(fg) \leq \lceil log_3 f(fg) \rceil$$

Donc $f(fc), f(fg), f(fd) \leq 3^k$ or :

$$f(T) = f(fc) + f(fg) + f(fd)$$

donc :

$$f(T) \leq 3 \times 3^k = 3^{k+1}$$

D' où on en déduit que :

$$h(T) \geq log_3 f(T)$$

Il y a $!n$ agencements boulons-écrous possibles donc l'arbre de décision, qui est ternaire a pour hauteur : $log_3 !n \sim n log_3 n$, donc la complexité dans le pire cas de l'algo est de : $\Omega(n \times log n)$.

Problème ouvert : proposer un algorithme en $o(n^2)$ essais pour résoudre ce problème.

(Star) **Exercice 3.** *Cherchez la Star !*

Dans un groupe de n personnes (numérotées de 1 à n pour les distinguer), une *star* est quelqu'un qui ne connaît personne mais que tous les autres connaissent. Pour démasquer une star, s'il en existe une, vous avez juste le droit de poser des questions à n'importe quel individu i du groupe, du type « est-ce que vous connaissez j ? ». On suppose que les individus répondent toujours la vérité. On veut un algorithme qui trouve une star s'il en existe une, et qui garantit qu'il n'y en a pas sinon, en posant le moins de questions possibles.

1. Combien peut-il y avoir de stars dans le groupe ?

   ☞ Il ne peut y avoir qu'une seule star dans le groupe, puisque s'il y en a une, elle ne connait personne, et donc tous les autres sont inconnus d'au moins une personne, et ne peuvent donc être eux aussi des stars.

2. Écrire le meilleur algorithme que vous pouvez et donner sa complexité en nombre de questions (on peut y arriver en $\mathcal{O}(n)$ questions).

   ☞ Lorsqu'on effectue le test $"i \to j\,?\,"$, c'est-à-dire lorsque l'on cherche à savoir si la personne i connaît la personne j, on obtient le résultat suivant :

   – si oui, alors i n'est pas une star, mais j en est potentiellement une.
   – si non, alors j n'est pas une star, mais i en est potentiellement une.

   L'algorithme consiste alors à parcourir le tableau des personnes une fois, en gardant en memoire à chaque instant la personne i qui est jusqu'ici reconnue par tous, tandis qu'au $j^{\text{ème}}$ test, toutes les autres personnes, dont les indices sont les $k < j$ (avec $k \neq i$, bien sûr) ne peuvent pas être des stars. Cet algorithme s'écrit donc de la manière suivante :

```
début
    i ← 1 et j ← 2;
    tant que j ≤ n faire
        si "j → i" alors j ← j + 1;
        sinon i ← j et j ← j + 1;
    i_star ← vrai et j ← 1;
    tant que j < i et i_star faire
        si "j → i" alors j ← j + 1;
        sinon i_star ← faux;
    k ← 1;
    tant que k ≤ n et i_star faire
        si k ≠ i et "i → k" alors i_star ← faux;
        sinon k ← k + 1;
    si i_star alors retourner "i est la star";
    sinon retourner "il n'y a pas de star";
```

3. Donner une borne inférieure sur la complexité (en nombre de questions) de tout algorithme résolvant le problème.
   (*Difficile*) Prouver que la complexité exacte de ce problème est $3n - \lfloor \log_2(n) \rfloor - 3$.

   ☞ Une borne inférieure facile est $(2n-2)$. En effet, pour s'assurer qu'il existe une star, on doit vérifier si tout le monde la connaît et qu'elle ne connaît personne. Cela demande donc deux tests par personnes autre que la star, soit $(2n-2)$.

   Nous allons montrer que la complexité exacte est $3n - \lfloor \log n \rfloor - 3$ en exhibant un algorithme ayant cette complexité, ainsi qu'en prouvant une borne inférieure. Pour cela, on introduit la notion de *match* : on voit les n personnes comme n joueurs, et un match entre i et j est simplement un test « $i \to j$? » ou « $j \to i$? ». Le *gagnant* d'un match est celui des deux qui peut toujours être star (donc c'est j si $i \to j$ et i sinon, et inversement pour le test $j \to i$). Lors d'un algorithme pour déterminer la présence d'une star, un joueur i est *éliminé* s'il perd un match, et qu'il n'en avait pas perdu avant.

   Algorithme. L'algorithme proposé possède deux phases. La première consiste en $(n-1)$ matches pour déterminer la star potentielle, et la deuxième vérifie si cette star potentielle est une vraie star. Le but du jeu est de faire en sorte que la star potentielle ait participé au plus de matches possibles dans la première phase pour que la seconde phase soit la plus courte possible.

   On voit la première phase comme un tournoi. À chaque tour, un certain nombre de matches ont lieu et certains joueurs sont éliminés. Le dernier tour (la finale) oppose deux joueurs et il ne reste qu'une star potentielle. À chaque tour, il peut arriver qu'un joueur ne joue pas, s'il y a un nombre impair de joueurs. On dit qu'un tel joueur a eu un *repos*. On crée alors un tournoi ayant les propriétés suivantes : après chaque tour, chaque vainqueur a effectué au plus un repos depuis le début du tournoi, et au plus deux vainqueurs ont effectué un repos. Clairement, c'est facile de faire cela après le premier tour (si n est impair, un joueur effectue un repos et si n est pair, aucun joueur n'a eu de repos). Si c'est le cas après le tour t, il y a trois cas : si aucun vainqueur n'a eu de repos, on organise un nouveau tour arbitraire ; si exactement un vainqueur a eu un repos, on organise le tour en s'assurant que ce vainqueur joue bien à ce tour ; si deux vainqueurs ont eu un repos, ils jouent l'un contre l'autre et les autres matches sont arbitraires. On vérifie aisément qu'après ce nouveau tour, les propriétés sont toujours respectées. Puisque chaque match élimine exactement un joueur, il y a $(n-1)$ matches dans ce tournoi. De plus, d'après les propriétés du tournoi, la star potentielle a participé à tous les tours sauf au plus un. Comme il y a $\lceil \log n \rceil$ tours, la star potentielle a participé à au moins $\lfloor \log n \rfloor$ matches ($\lceil \log n \rceil = \lfloor \log n \rfloor$ si et seulement si n est une puissance de deux, auquel cas les joureurs n'ont jamais de repos).

Notons s la star potentielle repérée lors de la première phase. Pour la deuxième phase, il suffit d'effectuer tous les tests « $s \to i$? » et « $i \to s$? » non encore effectués. D'après la remarque précédente, il y en a au plus $2n - 2 - \lfloor \log n \rfloor$. L'algorithme effectue donc le nombre de tests que l'on souhaitait.

Borne inférieure. On rappelle qu'on appelle *élimination* un match au cours duquel un joueur qui n'avait jamais perdu est éliminé. Il est facile de voir que lors tout algorithme résolvant le problème de la star, au moins $(n-1)$ éliminations ont lieu. En effet, s'il y a une star, les $(n-1)$ autres joueurs doivent avoir été éliminés et s'il n'y en a pas, tous les joueurs doivent être éliminés (soit n éliminations). D'autre part, s'il y a une star, elle doit avoir participé à $(2n-2)$ matches. Il s'agit maintenant de montrer que dans le pire cas, le nombre d'éliminations auxquelles a participé la star est au plus $\lfloor \log n \rfloor$. Ainsi, le nombre total de matches étant supérieur à (nombre d'éliminations) + (nombre de matches de la star) - (nombre d'éliminations de la star), on obtient la borne inférieure souhaitée. Dans la suite, on appelle *perdant* un joueur qui a été éliminé, et *gagnant* un joueur non encore éliminé.

Pour cela, on construit un adversaire. Cet adversaire choisit au fur et à mesure les résultats des matches. Si on effectue deux fois le même test, ce qui n'est pas recommandé, l'adversaire doit donner deux fois le même résultat et c'est la seule règle qu'il doit respecter. La stratégie de l'adversaire est la suivante. Pour un match entre deux perdants, le résultat est quelconque. Pour un match entre un perdant et un gagnant, l'adversaire fait gagner le gagnant. Pour un match entre deux gagnants, l'adversaire fait gagner celui qui a éliminé le moins de gens jusqu'ici. Il est clair qu'avec cette stratégie, il y aura une star à la fin.

On cherche maintenant à prouver qu'avec cette stratégie, la star aura éliminé au plus $\lfloor \log n \rfloor$ joueurs au total. Construisons un arbre T avec n sommets (représentant les n joueurs), et un arc $i \to j$ si i a éliminé j au cours de l'algorithme. On montre par induction que pour tout sous-arbre $S \subseteq T$ dont la racine a m fils, $|S| \geq 2^m$. Si la racine de S n'a pas de fils, $|S| = 1$. Sinon soit r la racine de S et notons $r_1, \ldots, r_m$ les fils de r et $S_1, \ldots, S_m$ les sous-arbres de S dont les racines sont $r_1, \ldots, r_m$, respectivement. Par définition, r a éliminé $r_1, \ldots, r_m$ durant l'algorithme. Supposons que r_1 fût le premier éliminé, puis r_2, etc... Quand r élimine r_j (où $1 \leq j \leq m$), r_j doit avoir éliminé plus de joueurs que r d'après la stratégie de l'adversaire. Or quand r élimine r_j, il a éliminé exactement $(j-1)$ joueurs auparavant. Ainsi, r_j en a éliminé au moins autant et dans S, r_j a au moins $(j-1)$ fils. Par hypothèse d'induction, $|S_j| \geq 2^{j-1}$. Ainsi, $|S| \geq 1 + \sum_j |S_j| = 2^m$. Mais puisque S contient exactement n sommets, on a $n \geq 2^m$, c'est-à-dire $m \leq \lfloor \log n \rfloor$. Ceci prouve que lors de l'algorithme, la star aura éliminé au plus $\lfloor \log n \rfloor$ joueurs. Ceci suffit à conclure.