Bornes inférieures de complexité I

Cours aux Journées ALÉA 2020

Cyril Nicaud

LIGM – Univ Gustave Eiffel & CNRS

Mars 2020

Philippe Flajolet ... 10 ans déjà

Avant propos

- C'est un cours d'algorithmique, on s'attardera plus sur les idées et les techniques de preuves que sur le formalisme mathématique
- Je ne suis pas un spécialiste du sujet (je l'enseigne au niveau M1, mais on va aller plus loin)
- Le cours est organisé ... comme un cours, en prenant son temps

- Il y a assez peu de ressources sur le sujet (hors articles de recherche). On se réfèrera à
  - La page de Jeff Erickson avec ses notes de cours sur les bornes inférieures par comptage et sur les arguments d'adversaire : `http://web.engr.illinois.edu/~jeffe/teaching/algorithms/notes/`
  - Le livre "Randomized Algorithms" de R. Motwani & P. Raghavan, pour les bornes inférieures d'algorithmes probabilistes

C'est parti !

1. Complexité des algorithmes et des problèmes

Complexité d'un algorithme (principe)

Buts: estimation de performances / comparaison d'algorithmes

```
def search(x,T):
  for y in T:
    if x == y:
      return True
  return False
```

- E_n les entrées de taille n, et $E = \cup_n E_n$
- $C(e)$ le nombre d'instructions effectuées pour l'entrée e
- C est une application de E dans $\mathbb{N}$

Comment estimer / comparer deux applications de $E \longrightarrow \mathbb{N}$?

- On agrège l'information à taille n fixée
- $C_n = \max\{C(e) : e \in E_n\}$ (pire cas) ou $C_n = \mathbb{E}_{E_n}[C]$ (cas moyen)

Comment estimer / comparer deux suites de $\mathbb{N} \longrightarrow \mathbb{N}$?

- On les estime asymptotiquement
- On utilise les notations $\mathcal{O}$, Ω et Θ

$u_n = \mathcal{O}(v_n)$ quand il existe $c > 0$ tq $u_n \leq cv_n$ à partir d'un certain rang
$u_n = \Omega(v_n)$ quand il existe $c > 0$ tq $u_n \geq cv_n$ à partir d'un certain rang
$u_n = \Theta(v_n)$ quand $u_n = \mathcal{O}(v_n)$ et $u_n = \Omega(v_n)$

Complexité d'un algorithme (exemples)

```
def search(x,T):
  for y in T:
    if x == y:
      return True
  return False
```

- Pire cas : $\Theta(n)$
- Cas moyen : $\Theta(n)$

```
def qsort(x,T):
  if len(T) <= 1: return T
  G = [x in T[1:] if x<T[0]]
  D = [x in T[1:] if x>=T[0]]
  return qsort(G)+[T[0]]+qsort(D)
```

- Pire cas : $\Theta(n^2)$
- Cas moyen : $\Theta(n \log n)$

```
def mergesort(x,T):
  if len(T) <= 1: return T
  G = mergesort(T[:n//2])
  D = mergesort(T[n//2:])
  return merge(G,D)
```

- Pire cas : $\Theta(n \log n)$
- Cas moyen : $\Theta(n \log n)$

Complexité d'un problème (définition)

"Definition" : un problème est une tâche à effectuer par ordinateur (une application de E dans un ensemble F).

Exemples : trier un tableau (**SORT**), tester s'il y a trois éléments x, y, z dans T tels que $x + y + z = 0$ (**3SUM**)

Un même problème peut être résolu par plusieurs algorithmes.

Définition (complexité d'un problème)

Un problème Π est de complexité $\mathcal{O}(u_n)$ quand il existe un algorithme de complexité $\mathcal{O}(u_n)$ qui résout Π.

Un problème Π est de complexité $\Omega(u_n)$ quand tout algorithme qui résout Π est de complexité $\Omega(u_n)$: on dit alors que $\Omega(u_n)$ est une borne inférieure de complexité pour Π.

Le problème **SORT** est de complexité $\mathcal{O}(n \log n)$, car il est résolu par l'algorithme `mergesort` qui est en $\mathcal{O}(n \log n)$ (pire cas).

Complexité d'un problème (modèle de calcul)

Définition (complexité d'un problème)

Un problème Π est de complexité $\Omega(u_n)$ quand tout algorithme qui résout Π est de complexité $\Omega(u_n)$: on dit alors que $\Omega(u_n)$ est une borne inférieure de complexité pour Π.

```
def search(x,T):
  for y in T:
    if x == y:
      return True
  return False
```

Pour estimer la complexité d'un **algorithme**, il suffit de définir ce que coûtent ses différentes instructions : ici en temps $\mathcal{O}(1)$: aller au y suivant, tester si $x == y$, ...

Pour les bornes inférieures de complexité d'un **problème** :

- Il faut dire quelque-chose sur tous les algorithmes !
- Il faut donc définir **précisément** ce qu'est un algorithme : il faut définir un **modèle d'ordinateur** (machine de Turing, RAM, ...)

Complexité d'un problème (exemple)

Rappel : SORT est le problème de trier un tableau de taille n

Exemple de théorème (tri par comparaisons)

Dans le modèle de calcul où l'on peut juste comparer les données d'un tableau, le problème **SORT** est en $\Omega(n \log n)$.

Seul accès aux données : "Est-ce que $T[i] < T[j]$?"

- ☹ Le résultat ne porte que sur les **tris par compaisons**
- ☺ Pas besoin de spécifier le modèle d'ordinateur précisément
- ☺ Pas besoin de spécifier l'encodage des données
- ☺ L'algorithme `mergesort` est optimal pour ce modèle

Dans la suite on utilisera ce type de modèles car il est très difficile d'obtenir des résultats non-triviaux dans des modèles généraux : RAM, machines de Turing (*pas réaliste pour des pb polynomiaux*), ...

Borne inférieure d'un problème (récapitulatif)

Définition (complexité d'un problème)

Un problème Π est de complexité $\Omega(u_n)$ quand tout algorithme qui résout Π est de complexité $\Omega(u_n)$: on dit alors que $\Omega(u_n)$ est une borne inférieure de complexité pour Π.

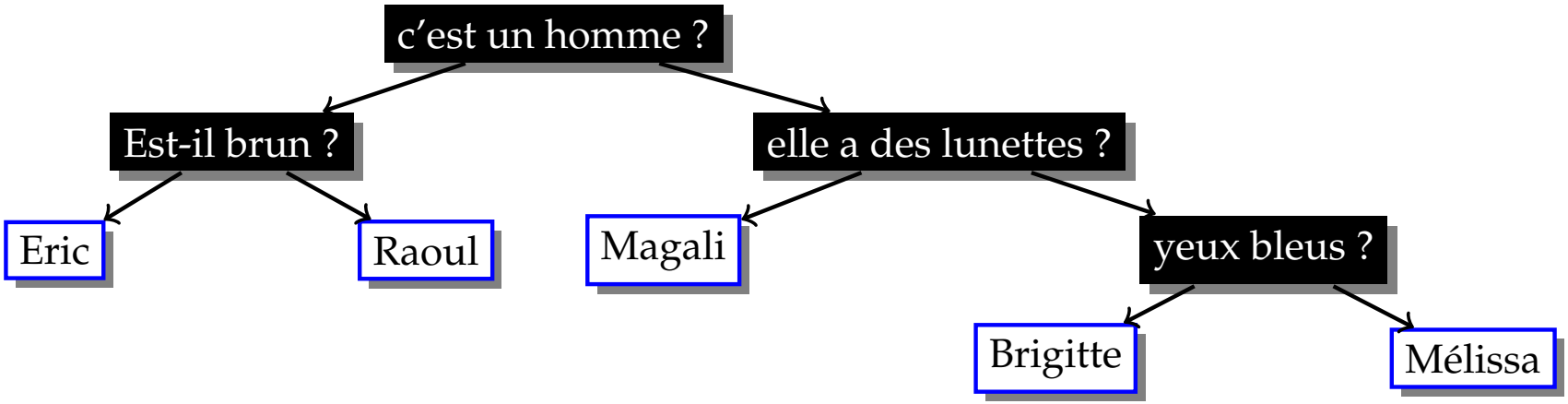
- Ce n'est pas (du tout) une complexité meilleur cas d'un algo
- On doit spécifier un modèle de calcul pour pouvoir dire quelque chose sur **tous** les algorithmes qui sont solution du problème
- On va s'intéresser à des problèmes polynomiaux : pas à des questions de type **NP**-difficulté (qui sont aussi des bornes inf.)
- On va utiliser des modèles simples de calcul, et passer du temps à interpréter les résultats obtenus

2. Comptage

Le jeu “Qui est-ce ?”

- Chaque joueur tire au sort une carte personnage
- Ils posent des **questions oui/non** tour à tour pour identifier le personnage tiré par l’autre joueur

Modèle "arbre de décision"



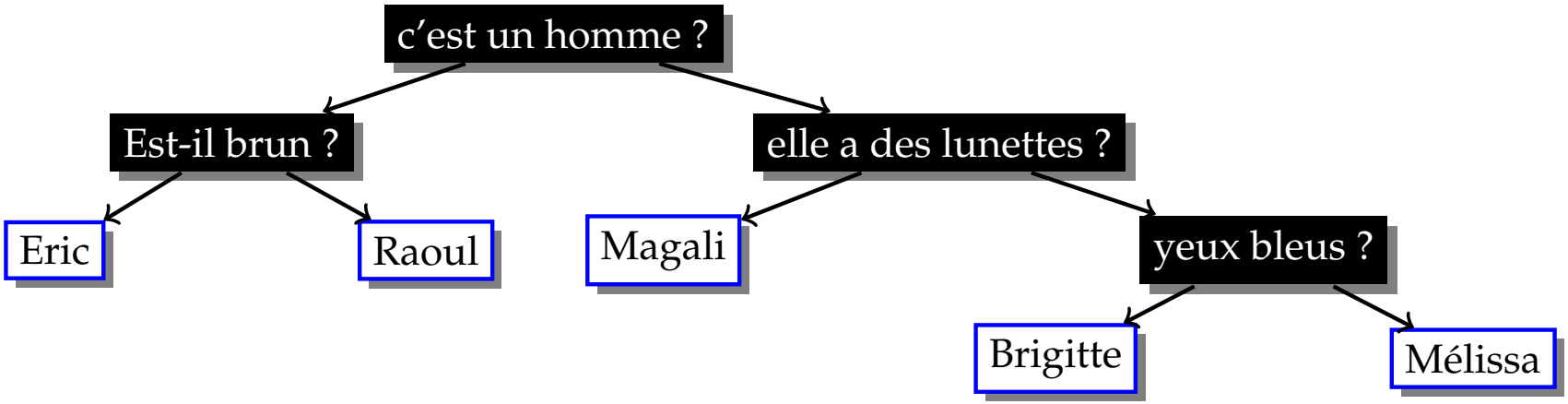
Dans le *modèle "arbre de décision" (MAD)* :

- Pour chaque n, on a un algorithme sous forme d'**un arbre de questions oui/non** pour identifier la réponse
- Le coût d'une exécution est le **nombre de questions** posées
- La complexité pire cas est la **hauteur de l'arbre**

Sur l'exemple, la complexité au pire est de **3** (*Brigitte* et *Mélissa*).

C'est un modèle simple où sont négligés tous les calculs qui ne sont pas des questions $\Rightarrow$ c'est OK car on cherche des bornes inférieures

La technique par comptage



Tous les algos pour résoudre le problème $=$ tous les arbres de décision

Théorème (argument de comptage)

Dans le **modèle "arbre de décision"**, si tout algorithme qui résout un problème doit produire au moins R_n réponses différentes pour les entrées de taille n alors le problème admet une borne inférieure de complexité en $\log_2 R_n$.

Preuve : un arbre binaire de hauteur h a au plus 2^h feuilles □

Le problème SORT (définition)

"Definition :" étant donné un tableau de n nombres, **SORT** consiste à le ranger par ordre croissant

☹ dans un MAD, on n'a pas accès directement aux valeurs stockées dans le tableau $\Rightarrow$ on ne peut pas retourner le tableau trié

Définition (SORT)

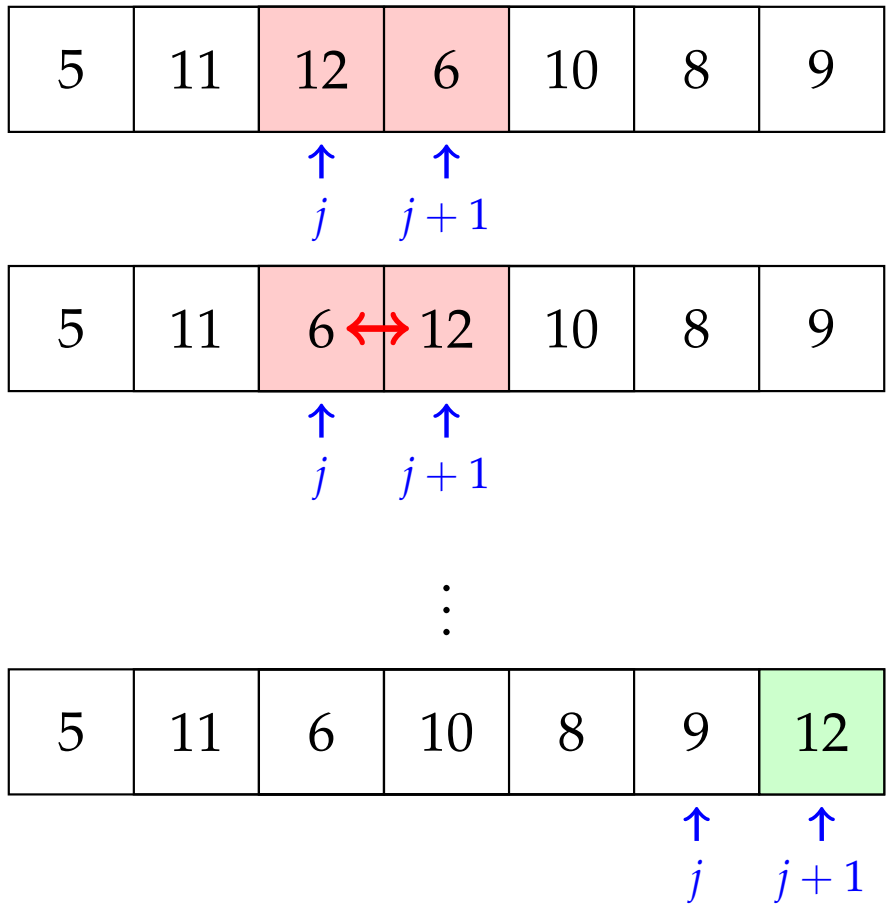
Soit T un tableau de n nombres, le problème **SORT** consiste à construire une permutation σ de $\{0 \dots n-1\}$ telle que

$$T[\sigma(0)] \leq T[\sigma(1)] \leq \dots \leq T[\sigma(n-1)]$$

c'est-à-dire de trouver comment permuter les éléments de T de sorte qu'ils soient dans l'ordre croissant.

Le problème SORT (tri bulle, présentation)

```
def triBulle(x,T):
  for i in range(len(T)):
    for j in range(len(T)-i-1):
      if T[j] > T[j+1]:
        T[j],T[j+1] = T[j+1],T[j]
  return T
```



Le problème SORT (tri bulle, reformulation)

```
def triBulle(T):
  for i in range(len(T)):
    for j in range(len(T)-i-1):
      if T[j] > T[j+1]:
        T[j],T[j+1] = T[j+1],T[j]
  return T
```

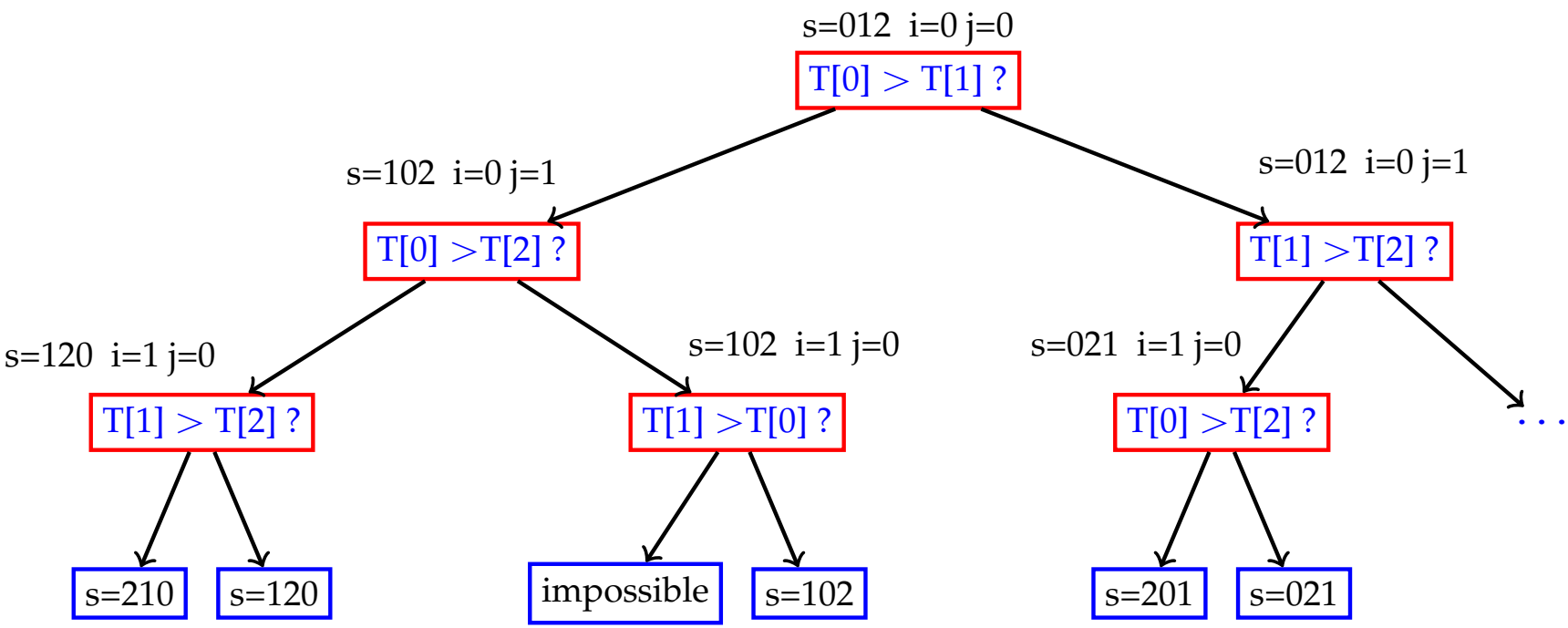
Est reformulé pour rentrer dans notre spécification (calculer une permutation qui ordonne les éléments) en :

```
def triBulle(T):
  s = list(range(len(T))) # s = perm. identité
  for i in range(len(T)):
    for j in range(len(T)-i-1):
      if T[s[j]] > T[s[j+1]]:
        s[j],s[j+1] = s[j+1],s[j]
  return s
```

On ne pose que des questions : “Est-ce que $T[i] < T[j]$?”
$\Rightarrow$ c’est un **tri par comparaisons**, donc dans le cadre MAD

Le tri bulle comme arbre de décision ($n = 3$)

```
def triBulle(T):
  for i in range(len(T)):
    for j in range(len(T)-i-1):
      if T[j] > T[j+1]:
        T[j],T[j+1] = T[j+1],T[j]
  return T
```



Borne inférieure pour SORT

Théorème (borne inf. SORT)

Dans le **modèle "arbre de décision"**, **SORT** admet une borne inférieure de complexité en $\Omega(n \log n)$.

Preuve : par comptage, car $\log n! \in \Omega(n \log n)$ □

Corollaire (borne inf. tris par comparaisons)

Tout **tri par comparaisons** nécessite $\Omega(n \log n)$ comparaisons dans le pire cas.

Attention

Théorème (argument de comptage)

Dans le **modèle "arbre de décision"**, si tout algorithme qui résout un problème doit produire au moins R_n réponses différentes pour les entrées de taille n alors le problème admet une borne inférieure de complexité en $\log_2 R_n$.

Si on considère le problème "Trouver deux indices $i \neq j$ tels que $T[i] \leq T[j]$", pour le modèle par comparaisons.

- Le problème admet tous les couples (i, j) comme solutions potentielles
- Mais un algorithme qui le résout peut ne répondre que $(0, 1)$ ou $(1, 0)$
- La borne inférieure par comptage est $\Omega(1)$!

```
def two_ordered(T):
  if T[0] <= T[1]:
    return (0,1)
  return (1,0)
```

Borne inférieure pour FIND_SORTED

Définition

Le problème **FIND_SORTED** consiste à chercher un élément x dans un tableau trié T de taille n. On doit retourner la position d'une occurrence de x si $x \in T$ et `False` sinon.

Remarque : une dichotomie résout le problème en $\mathcal{O}(\log n)$ comparaisons de x avec des éléments de T.

Théorème (borne inf. FIND_SORTED)

Dans le **modèle "arbre de décision"**, **FIND_SORTED** admet une borne inférieure de complexité en $\Omega(\log n)$.

Preuve : par comptage, car $\log_2(n+1) \in \Omega(\log n)$ □

Borne inférieure pour FIND

Définition

Le problème **FIND** consiste à chercher un élément x dans un tableau T de taille n (non nécessairement trié). On doit retourner la position d'une occurrence de x si $x \in T$ et `False` sinon.

Remarque : cette fois on ne peut pas faire une dichotomie.

Théorème (borne inf. FIND)

Dans le **modèle "arbre de décision"**, **FIND** admet une borne inférieure de complexité en $\Omega(\log n)$.

Preuve : par comptage, car $\log_2(n+1) \in \Omega(\log n)$ □

☹ La borne inférieure obtenue ne paraît pas très intéressante.

Borne supérieure pour FIND dans MAD

Théorème (borne inf. FIND)

Dans le **modèle "arbre de décision"**, **FIND** admet une borne inférieure de complexité en $\Omega(\log n)$.

En fait, dans le modèle MAD, on peut faire une dichotomie !

```
def dichoMAD(T, x):
  d, f = 0, len(T)-1
  while d <= f:
    m = (d+f) // 2
    if T[m] == x: #question MAD
      return m
    if "x est dans T entre les positions d et m-1": #question MAD
      f = m-1
    else:
      d = m+1
  return False
```

FIND : conclusion

Théorème (borne inf. FIND)

Dans le **modèle "arbre de décision"**, **FIND** admet une borne inférieure de complexité en $\Omega(\log n)$.

Théorème (borne sup. FIND)

Dans le **modèle "arbre de décision"**, **FIND** est résolu par un algorithme de complexité en $\mathcal{O}(\log n)$.

Conclusion :

- Le MAD est peu puissant car il n'autorise que les questions oui/non sur les données.
- Le MAD est très puissant car il traite ces questions en temps constant.

Technique par comptage : récapitulatif

Théorème (argument de comptage)

Dans le **modèle "arbre de décision"**, si tout algorithme qui résout un problème doit produire au moins R_n réponses pour les entrées de taille n alors le problème admet une borne inférieure de complexité en $\log_2 R_n$.

- Cela donne un outil pour obtenir des bornes inférieures
- On n'échappe pas à une discussion sur le modèle de calcul

Attention : un problème peut être résolu par des algorithmes qui ne retournent pas la même chose : il faut montrer que tout algorithme a au moins R_n réponses différentes.

Remarque : au lieu des questions binaires, on peut autoriser des k-aires, en changeant $\log_2$ en $\log_k$.

3. Adversaire

Exemple 1 : le jeu des couleurs

On considère le jeu (pas très fun) suivant :

- Alice choisit dans sa tête une couleur parmi $\{vert, blue, rouge, noir\}$
- Bob doit deviner la couleur en utilisant uniquement des questions de la forme : **"Est-ce que la couleur est x ?"**

Exemple 1 : le jeu des couleurs

On considère le jeu (pas très fun) suivant :

- Alice choisit dans sa tête une couleur parmi $\{vert, blue, rouge, noir\}$
- Bob doit deviner la couleur en utilisant uniquement des questions de la forme : **"Est-ce que la couleur est x ?"**

En trichant, mais sans que Bob puisse prouver quelle triche, Alice peut forcer Bob à poser au moins 4 questions avant de répondre **oui**.

Pour cela elle a une stratégie pour répondre aux questions, qui peut être décrite par l'algorithme :

- Initialement Alice note secrètement $\mathcal{C} = \{vert, blue, rouge, noir\}$
- À chaque fois que Bob demande si la couleur est x :
  - Si $\mathcal{C} = \{x\}$, Alice répond **oui**
  - Sinon, Alice répond **non** et enlève x de $\mathcal{C}$ (s'il y est)

On enlève au plus un élément par question et Bob ne peut pas prouver qu'Alice triche.

Exemple 2 : “Qui est-ce ?”

- Alice est une prestidigitatrice, elle peut changer sa carte à tout moment sans que Bob ne s’en aperçoive
- À chaque question de Bob, elle change éventuellement de carte pour que Bob élimine le moins de personnage possible.

Il faudra donc au moins $\log_2 n$ questions à Bob pour finir.

Preuve par adversaire : principe

- On veut montrer que tout algo nécessite au moins t étapes sur une entrée de taille n dans un certain modèle de calcul
- On donne les rôles suivants :
  - L'algorithme pose des questions et doit identifier la réponse à donner (il ne connaît l'entrée que par les questions qu'il a posées)
  - L'adversaire répond aux questions en tentant de faire durer le processus le plus longtemps possible
- On veut établir une stratégie pour l'adversaire telle que :
  - À chaque question, l'adversaire choisit une réponse telle qu'au moins une entrée satisfait ses réponses jusqu'ici (**cohérence**)
  - Après $t-1$ questions, il reste au moins deux entrées dont les résultats par l'algorithme sont différents

Remarque : en théorie il suffit de montrer l'existence d'une telle stratégie. En pratique, on la décrit souvent par un algorithme qui décrit les réponse à donner.

Retour sur FIND

Rappel : **FIND** consiste à déterminer si x est dans le tableau T

Modèle : On se place dans le modèle où l'algorithme peut poser comme question **"Quelle est la valeur de $T[i]$?"**

Stratégie de l'adversaire : répondre y à chaque fois, avec $y \neq x$

Théorème (FIND par adversaire)

Dans ce modèle, tout algorithme qui résout **FIND** utilise au moins n questions dans le pire cas.

Preuve : si l'algorithme a posé $n-1$ questions, il y a au moins une case dont il n'a pas demandé la valeur. L'adversaire peut encore choisir de placer x ou y dedans. □

Calcul du min et du max

Définition

Le problème **MIN_MAX** consiste à trouver le minimum et le maximum d'un tableau T (ou l'indice d'un minimum et l'indice d'un maximum).

Modèle : On se place dans le modèle **"par comparaisons"** où on peut poser des questions du type **"Est-ce que** $T[i] < T[j]$ **?"**

```
def naive_minmax(T):
  i_min, i_max = 0, 0
  for i in range(len(T)):
    if T[i] < T[i_min]:
      i_min = i
    if T[i] > T[i_max]:
      i_max = i
  return i_min, i_max
```

- On parcourt les indices du tableau
- On compare `T[i]` avec les min-records et max-records
- On met à jour `i_min` et `i_max`, si besoin

- $\sim 2n$ comparaisons pire cas

Min et Max : algorithme astucieux

- prendre les éléments deux par deux, les comparer entre eux
- comparer le plus petit au min courant
- comparer le plus grand au max courant

```
def minmax_group2(T):
  i_min, i_max = len(T)-1, len(T)-1
  for i in range(0,len(T)-1,2): # de 2 en 2
    if T[i] < T[i+1]:
      a, b = i, i+1
    else:
      a, b = i+1, i
    if T[a] < T[i_min]:
      i_min = a
    if T[b] > T[i_max]:
      i_max = b
  return i_min, i_max
```

Cela fait $\sim \frac{3}{2} n$ comparaisons.

Question : Peut-on faire mieux ?

Min et Max : borne inférieure (énoncé)

Rappel : **MIN_MAX** consiste à le min et le max de T

Modèle : accès aux données "par comparaisons", avec des questions du type **"Est-ce que $T[i] < T[j]$?"**

Théorème (borne inf. MINetMAX)

Dans le modèle **"par comparaisons"**, tout algorithme qui résout le problème **MIN_MAX** nécessite $\sim \frac{3}{2}n$ comparaisons dans le pire cas.

Remarque : l'algorithme astucieux `minmax_group2` est donc optimal

Min et Max : borne inférieure (preuve 1/2)

Alice remplit un tableau de taille n de symboles $\pm$. Les symboles s'interprètent de la façon suivante :

$\pm$	peut être le min ou le max	$+$	peut être le max
$-$	peut être le min	$\emptyset$	ni min ni max

Quand Bob pose une question (une comparaison) Alice donne la réponse qui enlève un minimum de signes ($\pm$ compte pour deux signes). Si Alice enlève le dernier symbole d'une case, elle y place un nombre compatible avec ses réponses antérieures.

tableau	question	réponse	signes perdus
0 1 2 3 4 $\pm$ $\pm$ $\pm$ $\pm$ $\pm$	$T[0] < T[2]$?	oui	2
0 1 2 3 4 $-$ $\pm$ $+$ $\pm$ $\pm$	$T[2] < T[4]$?	non	1
0 1 2 3 4 $-$ $\pm$ $+$ $\pm$ $-$	$T[0] < T[4]$?	oui	1
0 1 2 3 4 $-$ $\pm$ $+$ $\pm$ -1	...	...	...

Min et Max : borne inférieure (preuve 2/2)

Signes perdus :

- $\pm$ vs $\pm$: deux signes
- $\pm$ vs $+$ ou $\pm$ vs $-$: un signe
- $+$ vs $+$ ou $-$ vs $-$: un signe
- rien dans les autres cas

Remarques :

- Il y a toujours un signe $+$ et un signe $-$
- L'algo n'a pas terminé :
  - s'il y a un symbole $\pm$
  - s'il y a 3 signes

Objectif : minimiser $d + u$ avec

d = nombre de questions qui enlèvent deux signes ($\pm$ vs $\pm$)
u = nombre de questions qui enlèvent un signe

La meilleure stratégie est donc de poser un maximum de questions "$\pm$ vs $\pm$", soit $d = \lfloor n/2 \rfloor$, ce qui enlève $2\lfloor n/2 \rfloor$ signes

Comme il y a $2n$ signes initialement, il faut encore poser au moins $u = 2n - 2\lfloor n/2 \rfloor - 2$ questions qui enlèvent un signe

Au total cela fait $\lceil \frac{3n}{2} \rceil - 2 \approx \frac{3}{2}n$ questions

Remarque : si on n'avait pas l'algorithme astucieux, on pourrait le trouver à partir de l'analyse ci-dessus.

Conclusion sur les techniques

On a vu deux techniques :

- Par comptage
- Par adversaire

Qui sont des arguments **assez simples** (parfois astucieux) pour établir des bornes inférieures.

On a vu au passage qu'il est indispensable de préciser le modèle de calcul !

Plus d'exemples pendant la séance d'exercices !