Algorithmique Probabiliste

Philippe Duchon

LaBRI - ENSEIRB-Matméca - Université de Bordeaux

2014-15

Préambule : le schéma de Von Neumann

- **Question posée la semaine dernière :** on a une fonction `bern()` qui retourne des variables de Bernoulli indépendantes, toutes de paramètre p ($0 < p < 1$, inconnu) ; on souhaite écrire une fonction `flip()`, qui retourne une Bernoulli de paramètre $1/2$

Préambule : le schéma de Von Neumann

- **Question posée la semaine dernière :** on a une fonction `bern()` qui retourne des variables de Bernoulli indépendantes, toutes de paramètre p ($0 < p < 1$, inconnu) ; on souhaite écrire une fonction `flip()`, qui retourne une Bernoulli de paramètre $1/2$
- **Solution proposée :**

```
Repeter
    x = bern()
    y = bern()
Tant que x=y
Retourner x
```

Préambule : le schéma de Von Neumann

- **Question posée la semaine dernière :** on a une fonction `bern()` qui retourne des variables de Bernoulli indépendantes, toutes de paramètre p ($0 < p < 1$, inconnu) ; on souhaite écrire une fonction `flip()`, qui retourne une Bernoulli de paramètre $1/2$
- **Solution proposée :**

```
Repeter
    x = bern()
    y = bern()
Tant que x=y
Retourner x
```

- **On montre que :**
  - on termine avec probabilité 1
  - la probabilité de retourner 1 est égale à la probabilité de retourner 0 (donc à $1/2$)
  - le nombre (aléatoire) de passages dans la boucle est **géométrique**, d'espérance $1/2p(1-p)$

QuickSort en Python

```
def Partitionne(L,a,b,k):
    p = L[k]
    Echange(L,k,b)
    st = a
    for i in range(a,b):
        if L[i]<p:
            Echange(L,i,st)
            st = st+1
    Echange(L,st,b)
    return(st)

def QuickSortRec(L,a,b):
    if (a<b):
        k=Partitionne(L,a,b,a)
        QuickSortRec(L,a,k-1)
        QuickSortRec(L,k+1,b)
```

Analyse de **[Rand]QuickSort**

Théorème

Soit $n \geq 1$, $\sigma \in S_n$ une permutation quelconque de $[[1, n]]$, et soit

- QS_n, la variable aléatoire qui décrit le nombre de comparaisons de clés dans une exécution de l'algorithme déterministe **QuickSort** sur un *tableau de n valeurs distinctes, rangées initialement dans un ordre aléatoire uniforme*;
- $RQS_n(\sigma)$, la variable aléatoire qui décrit le nombre de comparaisons de clés dans une exécution de l'algorithme randomisé **RandQuickSort** sur un *tableau de n valeurs distinctes, rangées initialement selon l'ordre* σ.

Alors QS_n et $RQS_n(\sigma)$ ont la **même loi** : pour tout k,

$$\mathbb{P}(QS_n = k) = \mathbb{P}(RQS_n(\sigma) = k).$$

Analyse de **[Rand]QuickSort**

Théorème

Soit $n \geq 1$, $\sigma \in S_n$ une permutation quelconque de $[[1, n]]$, et soit

- QS_n, la variable aléatoire qui décrit le nombre de comparaisons de clés dans une exécution de l'algorithme déterministe **QuickSort** sur un *tableau de n valeurs distinctes, rangées initialement dans un ordre aléatoire uniforme*;
- $RQS_n(\sigma)$, la variable aléatoire qui décrit le nombre de comparaisons de clés dans une exécution de l'algorithme randomisé **RandQuickSort** sur un *tableau de n valeurs distinctes, rangées initialement selon l'ordre σ*.

Alors QS_n et $RQS_n(\sigma)$ ont la **même loi** : pour tout k,

$$\mathbb{P}(QS_n = k) = \mathbb{P}(RQS_n(\sigma) = k).$$

Corollaire 1 : même espérance, même variance...

Analyse de **[Rand]QuickSort**

Théorème

Soit $n \geq 1$, $\sigma \in S_n$ une permutation quelconque de $[[1, n]]$, et soit

- QS_n, la variable aléatoire qui décrit le nombre de comparaisons de clés dans une exécution de l'algorithme déterministe **QuickSort** sur un *tableau de n valeurs distinctes, rangées initialement dans un ordre aléatoire uniforme*;
- $RQS_n(\sigma)$, la variable aléatoire qui décrit le nombre de comparaisons de clés dans une exécution de l'algorithme randomisé **RandQuickSort** sur un *tableau de n valeurs distinctes, rangées initialement selon l'ordre* σ.

Alors QS_n et $RQS_n(\sigma)$ ont la **même loi** : pour tout k,

$$\mathbb{P}(QS_n = k) = \mathbb{P}(RQS_n(\sigma) = k).$$

Corollaire 1 : même espérance, même variance...

Corollaire 2 : pour **RandQuickSort**, il n'y a pas de tableau plus mauvais qu'un autre

Quelques commentaires. . .

- On a obtenu **RandQuickSort** à partir de **QuickSort** par randomisation : là où l'algorithme faisait un choix déterministe, mais arbitraire, on l'a remplacé par un choix aléatoire explicite

Quelques commentaires. . .

- On a obtenu **RandQuickSort** à partir de **QuickSort** par randomisation : là où l'algorithme faisait un choix déterministe, mais arbitraire, on l'a remplacé par un choix aléatoire explicite
- L'effet sur la complexité est un effet de **lissage** : **QuickSort** a de "bons" et de "mauvais" tableaux ; la plupart des tableaux sont "bons", mais si on examine le cas le pire, la possibilité de tomber sur un "mauvais" est gênante.

Quelques commentaires...

- On a obtenu **RandQuickSort** à partir de **QuickSort** par randomisation : là où l'algorithme faisait un choix déterministe, mais arbitraire, on l'a remplacé par un choix aléatoire explicite
- L'effet sur la complexité est un effet de **lissage** : **QuickSort** a de “bons” et de “mauvais” tableaux ; la plupart des tableaux sont “bons”, mais si on examine le cas le pire, la possibilité de tomber sur un “mauvais” est gênante.
- Le théorème nous dit que **RandQuickSort** se comporte sur tous les tableaux exactement comme **QuickSort** sur un tableau aléatoire (dans un ordre uniforme)

Quelques commentaires. . .

- On a obtenu **RandQuickSort** à partir de **QuickSort** par randomisation : là où l'algorithme faisait un choix déterministe, mais arbitraire, on l'a remplacé par un choix aléatoire explicite
- L'effet sur la complexité est un effet de **lissage** : **QuickSort** a de "bons" et de "mauvais" tableaux ; la plupart des tableaux sont "bons", mais si on examine le cas le pire, la possibilité de tomber sur un "mauvais" est gênante.
- Le théorème nous dit que **RandQuickSort** se comporte sur tous les tableaux exactement comme **QuickSort** sur un tableau aléatoire (dans un ordre uniforme)
- C'est souvent un objectif de la randomisation que de "casser les mauvaises instances"

Preuve (partielle) du théorème

- On définit $p_{k,n,\sigma}$: probabilité que **RandQuickSort**, sur une liste de n valeurs réparties selon σ, utilise exactement k comparaisons ($n \in \mathbb{N}$, $k \in \mathbb{N}$, $\sigma \in S_n$).
- On prouve des **récurrences** caractérisant toutes les $p_{k,n,\sigma}$
- Ces récurrences ne font pas intervenir σ : donc $p_{k,n,\sigma} = p_{k,n,\sigma'}$.

Récurrence

Hypothèse de récurrence H_n

Pour tout entier $m \leq n$, tout entier k, et toutes permutation σ et $\sigma' \in S_m$, on a $p_{k,m,\sigma} = p_{k,m,\sigma'}$.

Récurrence

Hypothèse de récurrence H_n

Pour tout entier $m \leq n$, tout entier k, et toutes permutation σ et $\sigma' \in S_m$, on a $p_{k,m,\sigma} = p_{k,m,\sigma'}$.

H_0, H_1 : S_0 et S_1 sont réduits à 1 élément, donc il n'y a rien à démontrer (au passage : $p_{k,0,()} = p_{k,1,(1)} = \delta_{k,0}$)

Récurrence

Hypothèse de récurrence H_n

Pour tout entier $m \leq n$, tout entier k, et toutes permutation σ et $\sigma' \in S_m$, on a $p_{k,m,\sigma} = p_{k,m,\sigma'}$.

H_0, H_1 : S_0 et S_1 sont réduits à 1 élément, donc il n'y a rien à démontrer (au passage : $p_{k,0,()} = p_{k,1,(1)} = \delta_{k,0}$)
Soit $n \geq 1$ tel que H_n soit vraie : on doit montrer H_{n+1}. Le seul entier $m \leq n+1$ qui n'est pas "couvert" par H_n est $m = n+1$, donc on va considérer une liste de longueur $n+1$.

Récurrence (suite)

- On considère une σ-liste L (de longueur $n+1$), et un entier k, et on se pose la question : à quelle condition est-ce que **RandQuickSort** va mettre exactement k comparaisons pour trier L ?

Récurrence (suite)

- On considère une σ-liste L (de longueur $n+1$), et un entier k, et on se pose la question : à quelle condition est-ce que **RandQuickSort** va mettre exactement k comparaisons pour trier L ?
- La partie non récursive de l'algorithme va faire exactement n comparaisons.

Récurrence (suite)

- On considère une σ-liste L (de longueur $n+1$), et un entier k, et on se pose la question : à quelle condition est-ce que **RandQuickSort** va mettre exactement k comparaisons pour trier L ?
- La partie non récursive de l'algorithme va faire exactement n comparaisons.
- On introduit la variable aléatoire $R \in [[1, n+1]]$: *rang* dans L de l'élément choisi comme pivot.

Récurrence (suite)

- On considère une σ-liste L (de longueur $n+1$), et un entier k, et on se pose la question : à quelle condition est-ce que **RandQuickSort** va mettre exactement k comparaisons pour trier L ?
- La partie non récursive de l'algorithme va faire exactement n comparaisons.
- On introduit la variable aléatoire $R \in [[1, n+1]]$: *rang* dans L de l'élément choisi comme pivot.
- R est **uniforme sur** $[[1, n+1]]$ (à cause du choix d'un indice par `randint(a,b)`), et ceci ne dépend pas de σ

Récurrence (suite)

- On considère une σ-liste L (de longueur $n+1$), et un entier k, et on se pose la question : à quelle condition est-ce que **RandQuickSort** va mettre exactement k comparaisons pour trier L ?
- La partie non récursive de l'algorithme va faire exactement n comparaisons.
- On introduit la variable aléatoire $R \in [[1, n+1]]$: *rang* dans L de l'élément choisi comme pivot.
- R est **uniforme sur** $[[1, n+1]]$ (à cause du choix d'un indice par `randint(a,b)`), et ceci ne dépend pas de σ
- Connaissant (conditionnellement à) la valeur de R, on connaît la taille des deux sous-listes à trier récursivement : $R-1$ et $n+1-R$

Récurrence (suite)

- On considère une σ-liste L (de longueur $n+1$), et un entier k, et on se pose la question : à quelle condition est-ce que **RandQuickSort** va mettre exactement k comparaisons pour trier L ?
- La partie non récursive de l'algorithme va faire exactement n comparaisons.
- On introduit la variable aléatoire $R \in [[1, n+1]]$: *rang* dans L de l'élément choisi comme pivot.
- R est **uniforme sur** $[[1, n+1]]$ (à cause du choix d'un indice par `randint(a,b)`), et ceci ne dépend pas de σ
- Connaissant (conditionnellement à) la valeur de R, on connaît la taille des deux sous-listes à trier récursivement : $R-1$ et $n+1-R$
- Quelle que soit la valeur de R, $R-1$ et $n+1-R$ sont compris entre 1 et n, donc on peut leur appliquer H_n...

Récurrence (suite)

- On considère une σ-liste L (de longueur $n+1$), et un entier k, et on se pose la question : à quelle condition est-ce que **RandQuickSort** va mettre exactement k comparaisons pour trier L ?
- La partie non récursive de l'algorithme va faire exactement n comparaisons.
- On introduit la variable aléatoire $R \in [[1, n+1]]$: *rang* dans L de l'élément choisi comme pivot.
- R est **uniforme sur** $[[1, n+1]]$ (à cause du choix d'un indice par `randint(a,b)`), et ceci ne dépend pas de σ
- Connaissant (conditionnellement à) la valeur de R, on connaît la taille des deux sous-listes à trier récursivement : $R-1$ et $n+1-R$
- Quelle que soit la valeur de R, $R-1$ et $n+1-R$ sont compris entre 1 et n, donc on peut leur appliquer H_n...
- (Reste à faire les calculs)

Récurrence (suite)

On définit quelques événements :

- $\mathcal{R}_k$: la liste L est triée en k comparaisons

Récurrence (suite)

On définit quelques événements :

- $\mathcal{R}_k$: la liste L est triée en k comparaisons
- $\mathcal{P}_r = \{R = r\}$

Récurrence (suite)

On définit quelques événements :

- $\mathcal{R}_k$: la liste L est triée en k comparaisons
- $\mathcal{P}_r = \{R = r\}$
- $\mathcal{Q}_i$: la liste $L_<$ est triée en i comparaisons

Récurrence (suite)

On définit quelques événements :

- $\mathcal{R}_k$: la liste L est triée en k comparaisons
- $\mathcal{P}_r = \{R = r\}$
- $\mathcal{Q}_i$: la liste $L_<$ est triée en i comparaisons
- $\mathcal{Q}'_j$: la liste $L_>$ est triée en j comparaisons

Récurrence (suite)

On définit quelques événements :

- $\mathcal{R}_k$: la liste L est triée en k comparaisons
- $\mathcal{P}_r = \{R = r\}$
- $\mathcal{Q}_i$: la liste $L_<$ est triée en i comparaisons
- $\mathcal{Q}'_j$: la liste $L_>$ est triée en j comparaisons

$$\mathcal{R}_k = \bigcup_{r=1}^{n+1} \bigcup_{i+j+n=k} \mathcal{P}_r \cap \mathcal{Q}_i \cap \mathcal{Q}'_j$$

Récurrence (suite)

On définit quelques événements :

- $\mathcal{R}_k$: la liste L est triée en k comparaisons
- $\mathcal{P}_r = \{R = r\}$
- $\mathcal{Q}_i$: la liste $L_<$ est triée en i comparaisons
- $\mathcal{Q}'_j$: la liste $L_>$ est triée en j comparaisons

$$\mathcal{R}_k = \bigcup_{r=1}^{n+1} \bigcup_{i+j+n=k} \mathcal{P}_r \cap \mathcal{Q}_i \cap \mathcal{Q}'_j$$

$$\mathbb{P}(\mathcal{R}_k) = \sum_{r=1}^{n+1} \mathbb{P}(\mathcal{P}_r) \sum_{i+j+n=k} \mathbb{P}(\mathcal{Q}_i|\mathcal{P}_r).\mathbb{P}(\mathcal{Q}'_j|\mathcal{P}_r)$$

Récurrence (suite)

On définit quelques événements :

- $\mathcal{R}_k$: la liste L est triée en k comparaisons
- $\mathcal{P}_r = \{R = r\}$
- $\mathcal{Q}_i$: la liste $L_<$ est triée en i comparaisons
- $\mathcal{Q}'_j$: la liste $L_>$ est triée en j comparaisons

$$\mathcal{R}_k = \bigcup_{r=1}^{n+1} \bigcup_{i+j+n=k} \mathcal{P}_r \cap \mathcal{Q}_i \cap \mathcal{Q}'_j$$

$$\mathbb{P}(\mathcal{R}_k) = \sum_{r=1}^{n+1} \mathbb{P}(\mathcal{P}_r) \sum_{i+j+n=k} \mathbb{P}(\mathcal{Q}_i|\mathcal{P}_r).\mathbb{P}(\mathcal{Q}'_j|\mathcal{P}_r)$$

On applique l'hypothèse de récurrence :

$$\mathbb{P}(\mathcal{Q}_i|\mathcal{P}_k) = p_{r-1,i} \text{ et } \mathbb{P}(\mathcal{Q}'_j|\mathcal{P}_k) = p_{n+1-k,j}$$

Au passage...

- **Arbre d'exécution de RandQuickSort** : arbre binaire,
  - racine : le pivot choisi
  - sous-arbre gauche : l'arbre d'exécution sur la liste $L_<$
  - sous-arbre droit : l'arbre d'exécution sur la liste $L_>$

Au passage. . .

- **Arbre d'exécution de RandQuickSort** : arbre binaire,
  - racine : le pivot choisi
  - sous-arbre gauche : l'arbre d'exécution sur la liste $L_<$
  - sous-arbre droit : l'arbre d'exécution sur la liste $L_>$
- Par construction, **c'est un arbre binaire de recherche** (toute clé qui se trouve dans le sous-arbre gauche [droit] d'un noeud, est inférieure [supérieure] à la clé de ce noeud)

Au passage...

- **Arbre d'exécution de RandQuickSort** : arbre binaire,
  - racine : le pivot choisi
  - sous-arbre gauche : l'arbre d'exécution sur la liste $L_<$
  - sous-arbre droit : l'arbre d'exécution sur la liste $L_>$
- Par construction, **c'est un arbre binaire de recherche** (toute clé qui se trouve dans le sous-arbre gauche [droit] d'un noeud, est inférieure [supérieure] à la clé de ce noeud)
- En termes de probabilités :
  - le noeud racine est uniformément choisi parmi les n clés
  - conditionnellement à leurs tailles (déterminées par le rang de la racine), les deux sous-arbres sont **indépendants** et **distribués comme des arbres d'exécution aléatoires de RandQuickSort**

Au passage...

- **Arbre d'exécution de RandQuickSort** : arbre binaire,
  - racine : le pivot choisi
  - sous-arbre gauche : l'arbre d'exécution sur la liste $L_<$
  - sous-arbre droit : l'arbre d'exécution sur la liste $L_>$
- Par construction, **c'est un arbre binaire de recherche** (toute clé qui se trouve dans le sous-arbre gauche [droit] d'un noeud, est inférieure [supérieure] à la clé de ce noeud)
- En termes de probabilités :
  - le noeud racine est uniformément choisi parmi les n clés
  - conditionnellement à leurs tailles (déterminées par le rang de la racine), les deux sous-arbres sont **indépendants** et **distribués comme des arbres d'exécution aléatoires de RandQuickSort**
- Ceci **caractérise** la loi : *toute distribution de probabilité sur les arbres binaires de taille n qui satisfait cette relation, est identique à celle des arbres d'exécution de* **RandQuickSort**.

Complexité moyenne de **RandQuickSort**

- On sait que la loi du nombre de comparaisons ne dépend que de la taille de la liste (et pas de son ordre original)

Complexité moyenne de **RandQuickSort**

- On sait que la loi du nombre de comparaisons ne dépend que de la taille de la liste (et pas de son ordre original)
- On s'intéresse donc à l'**espérance** (et autres grandeurs) de RQS_n

Complexité moyenne de **RandQuickSort**

- On sait que la loi du nombre de comparaisons ne dépend que de la taille de la liste (et pas de son ordre original)
- On s'intéresse donc à l'**espérance** (et autres grandeurs) de RQS_n
- On va donner une formule **exacte** pour $\mathbb{E}(RQS_n)$.

Complexité moyenne de **RandQuickSort**

- On sait que la loi du nombre de comparaisons ne dépend que de la taille de la liste (et pas de son ordre original)
- On s'intéresse donc à l'**espérance** (et autres grandeurs) de RQS_n
- On va donner une formule **exacte** pour $\mathbb{E}(RQS_n)$.
- Méthode : expression de RQS_n comme *somme de variables de Bernoulli* (non indépendantes), et *linéarité de l'espérance*

Décomposition de RQS_n

- $X_{i,j}$: variable aléatoire qui vaut 1 si les i-ème et j-ème clés (pour l'ordre croissant) sont comparées, 0 sinon.

Décomposition de RQS_n

- $X_{i,j}$: variable aléatoire qui vaut 1 si les i-ème et j-ème clés (pour l'ordre croissant) sont comparées, 0 sinon.
- $RQS_n = \sum_{1 \leq i < j \leq n} X_{i,j}$

Décomposition de RQS_n

- $X_{i,j}$: variable aléatoire qui vaut 1 si les i-ème et j-ème clés (pour l'ordre croissant) sont comparées, 0 sinon.
- $RQS_n = \sum_{1 \leq i < j \leq n} X_{i,j}$
- **Donc** $\mathbb{E}(RQS_n) = \sum_{1 \leq i < j \leq n} \mathbb{E}(X_{i,j})$

Décomposition de RQS_n

- $X_{i,j}$: variable aléatoire qui vaut 1 si les i-ème et j-ème clés (pour l'ordre croissant) sont comparées, 0 sinon.
- $RQS_n = \sum_{1 \leq i < j \leq n} X_{i,j}$
- **Donc** $\mathbb{E}(RQS_n) = \sum_{1 \leq i < j \leq n} \mathbb{E}(X_{i,j})$
- $X_{i,j}$ est une variable de Bernoulli : $\mathbb{E}(X_{i,j}) = \mathbb{P}(X_{i,j} = 1)$

Décomposition de RQS_n

- $X_{i,j}$: variable aléatoire qui vaut 1 si les i-ème et j-ème clés (pour l'ordre croissant) sont comparées, 0 sinon.
- $RQS_n = \sum_{1 \leq i < j \leq n} X_{i,j}$
- **Donc** $\mathbb{E}(RQS_n) = \sum_{1 \leq i < j \leq n} \mathbb{E}(X_{i,j})$
- $X_{i,j}$ est une variable de Bernoulli : $\mathbb{E}(X_{i,j}) = \mathbb{P}(X_{i,j} = 1)$
- Yapluka : trouver une expression pour $\mathbb{P}(X_{i,j} = 1)$, **et** faire les calculs.