TIPE

Conception des groupes dans la procédure du test groupé probabiliste de Dorfman

Aloïs Pouzin

Années 2021 – 2022

Table des matières

Première partie
Introduction

1. Principe

On cherche à tester la présence d'un caractère (une maladie par exemple) dans une population. Pour simplifier, on appellera toujours maladie le caractère à tester (bien que celui-ci peut être différent d'une maladie ; un trait héréditaire par exemple).

Tester tous les individus de la population peut s'avérer coûteux et long à réaliser, en particulier si la taille de cette population est grande.

Dorfman propose en 1943 une méthode permettant de diminuer le nombre de tests réalisés [1].

Définition 1.1 : Principe du test groupé de Dorfman

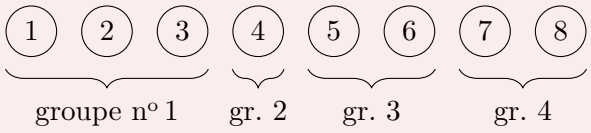
Le *group testing* ou *test groupé* de Dorfman consiste à réduire le nombre de tests effectués selon la procédure suivante [5] :

- on forme des groupes d'individus ;
- on teste le groupe dans son ensemble (et non chaque individu), et selon le résultat :
  - si le test est négatif, on considère le groupe, et donc chaque individu négatif ;
  - si le test est positif, on effectue des tests complémentaires individuels pour chaque individu du groupe.

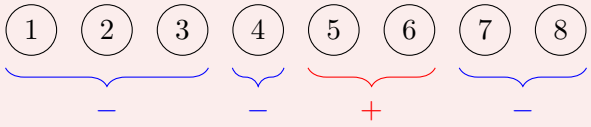
Depuis, d'autres méthodes de test groupé plus élaborées ont été développés [5]. On se limitera cependant ici à la procédure originelle de Dorfman, étant celle la plus utilisée de nos jours [5] [3].

Exemple 1.1 : cas où la procédure est efficace

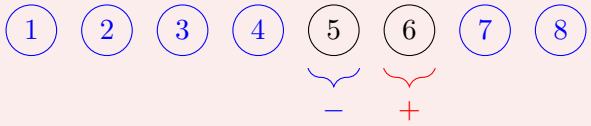
Considérons une population de 8 individus, et utilisons la procédure du test groupé de Dorfman. On forme arbitrairement des groupes suivant le schéma ci-dessous :



Si les résultats des tests sur les groupes sont suivants le schéma ci-contre (+ signifiant que le test est positif, et − signifiant que le test est négatif) :



alors on considère les individus 1, 2, 3, 4, 7 et 8 des groupes n^{os} 1, 2 et 4 sains ; et on réalise des tests complémentaires pour les autres sujets du groupe n^{o} 3. On peut obtenir par exemple :

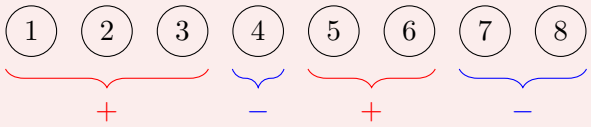


pour un résultat final :

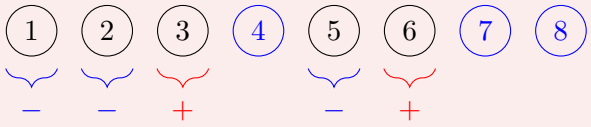
À l'issue de la procédure ont été réalisés 6 tests : c'est moins que si on avait testé toute la population.

Exemple 1.2 : cas où la procédure est inefficace

On reprend la même population que dans l'exemple 1.1.
Si les résultats des tests sur les groupes sont suivants le schéma ci-contre :



alors on considère les individus 4, 7 et 8 des groupes n^{os} 2 et 4 sains ; et on réalise des tests complémentaires pour les autres sujets des groupes n^{os} 1 et 3. Et si on obtient ensuite :



pour un résultat final :

on aura réalisé 9 tests, ce qui est plus que la taille de la population.

Remarque. Si un groupe ne contient qu'un individu, et si le test est positif, on ne réalise pas de test complémentaire. On considère directement le sujet comme malade.

On sera amené à faire cette disjonction de cas sur la taille des groupes plus tard.

On voit ainsi que dans certains cas, le processus n'est pas efficace, et que dans d'autres cas, le nombre de test est inférieur au nombre de sujets. Avec ces exemples, on peut intuiter que la procédure est plus efficace lorsque les groupes ont tendance à donner des résultats négatifs. Il vaudrait donc mieux utiliser le test groupé pour des individus ayant peu de chance d'avoir la maladie. On confimera cette intuition plus tard.

Remarque. Comment peut-on tester un groupe en une seule fois ? On peut par exemple prélever des échantillons chez chaque individu, mélanger ces prélèvements, puis tester le mélange. Généralement, le test prend beaucoup plus de temps et/ou est plus coûteux que les prélèvements [5].

On peut interroger la fiabilité de la procédure sur le mélange des échantillons (pourquoi peut on dire que tous les sujets d'un groupe testé négatif sont négatifs ?). On n'abordera pas cette question ici.

Comment alors choisir les groupes ? Comment faut-il former les groupes d'individus (taille des groupes, proximité des sujets...) pour que la procédure soit efficace ?

L'objet de ce TIPE est de trouver les conceptions idéales dans la procédure du test groupé.

2. Modélisation

Il existe deux modélisations possibles [2] :

— celle dite combinatoire qui consiste à étudier le pire des cas en se fixant le nombre maximal de malades possible dans la population ;

— celle dite probabiliste qui consiste à imposer une distribution de probabilité sur la positivité des sujets au test.

On utilisera l'approche probabiliste, considérée dans la littérature scientifique comme plus réaliste [2].

Définition 2.1

On note N le nombre de sujets, ou encore la taille de la population.

On suppose que N est suffisamment grand (on teste de grandes populations). On considèrera les événements : « un individu donné est malade » mutuellement indépendants.

Définition 2.2

On dit que la population est *homogène* lorsqu'elle est composée d'individus dont la probabilité d'être malade est la même.
On dit que la population est *hétérogène* lorsque cette probabilité est propre à chaque sujet.

On va distinguer ces deux cas par la suite.

2.1. Population homogène

Définition 2.3

On introduit $p \in [0;1]$ la probabilité qu'un individu quelconque de la population soit malade.

On n'étudiera pas les cas extrêmes $p = 0$ ou 1 qui rendent inutiles les tests.

Les sujets étant vus comme identiques par rapport au test de la maladie, il n'est pas nécessaire de distinguer et d'identifier chaque individu : on peut échanger deux sujets de deux groupes différents car aux yeux du test de la maladie, ce sont les mêmes individus, ayant la même probabilité d'être malade. En ce sens, donner une manière de former les groupes pour la procédure du test groupé revient seulement à donner le nombre d'individus qu'il y a dans chaque groupe. Il n'est pas nécessaire de savoir précisément quel sujet est dans quel groupe.

Définition 2.4

Une *partition d'un entier* naturel non nul est une décomposition de celui-ci en somme finie d'entiers naturels non nuls. Cette décomposition est à l'ordre près : l'ordre des entiers d'une décomposition n'importe pas.

Exemple 2.1

Les partitions de 4 sont :
- $1+1+1+1=4$;
- $2+1+1=4$;
- $2+2=4$;
- $3+1=4$;
- 4.

On en compte 5.

Si $a_1 + a_2 + \cdots + a_k = N$ est une partition d'un entier N, on la notera plus simplement $(a_1, a_2, \ldots, a_k)$ avec la convention de trier les éléments par ordre décroissant : $a_1 \geqslant a_2 \geqslant \cdots \geqslant a_k$.

Définition 2.5

Pour représenter une conception donnée des groupes, on introduit une partition de N, que l'on notera τ, telle que chaque terme de celle-ci correspond à la taille d'un groupe.

Remarque. Le nombre d'éléments de τ correspond alors au nombre de groupes. Par la suite, on notera ce nombre $|\tau|$.

Exemple 2.2

Pour $N = 12$, la partition $\tau = (4, 4, 3, 1)$ représente le cas où l'on a formé deux groupes de quatre, un groupe de trois, et un groupe d'une personne. Il a donc été formé $|\tau| = 4$ groupes.

Propriété 2.1

L'application de l'ensemble des conceptions des groupes dans l'ensemble des partitions de N qui à une conception donnée associe τ est bijective.

Démonstration.

— Soient deux conceptions notées 1 et 2. Supposons $\tau_1 = \tau_2$. Alors les deux conceptions ont le même nombre de groupes, et il y a autant de groupes d'une taille donnée dans l'une que dans l'autre (sinon on ne pourrait avoir $\tau_1 = \tau_2$). Nécessairement, les deux conceptions sont les mêmes, d'où l'injectivité.

— Soit τ une partition de N. Notons $\tau = (a_1, a_2, \ldots, a_k)$. Alors la conception qui correspond à un groupe de a_1 personnes, un groupe de a_2 personnes,..., un groupe de a_k personnes (qui est une conception valide car $a_1+a_2+\cdots+a_k = N$) a pour image τ, d'où la surjectivité. □

On pourra alors parler de partition de N pour représenter sans ambiguïté une conception donnée des groupes.

2.2. Population hétérogène

Cette fois-ci, il faut distinguer les individus de la population à tester, car leur probabilité d'être malade leur est propre.

On identifie chaque individu de la population par un entier compris entre 1 et N.

Définition 2.6

Soit $i \in [\![1; N]\!]$ un individu. On note $p_i \in [0; 1]$ la probabilité qu'il soit malade.

On ordonne les individus de tel sorte à avoir $p_1 \leqslant p_2 \leqslant \cdots \leqslant p_N$.

Il faut aussi un autre moyen plus précis qu'une partition de N (qui ne donne que la taille des groupes) pour représenter une conception donnée.

Rappelons au préalable la définition d'une partition.

Définition 2.7

Une *partition* G d'un ensemble E est une partie de $\mathscr{P}(E)$ vérifiant :

— $\forall g \in G,\ g \neq \emptyset$;

— $\bigcup_{g \in G} g = E$;

— $\forall g_1, g_2 \in G,\ g_1 \neq g_2 \Longrightarrow g_1 \cap g_2 = \emptyset$.
On appellera par la suite *groupe* un élément de G.

Définition 2.8

Pour une conception des groupes donnée, on lui associe une partition Γ de $[\![1; N]\!]$ telle que chaque groupe de Γ correspond aux groupes de la conception.

Exemple 2.3

Une partition de $[\![1; 6]\!]$ est $\{\{1\}; \{2; 4; 5\}; \{3; 6\}\}$. Elle correspond au cas où on a formé, pour une population de 6 individus, 3 groupes : un groupe avec un seul sujet (le premier) ; un groupe de trois avec le deuxième, le quatrième et le sixième ; et enfin un groupe de deux avec les sujets restants.

Propriété 2.2

L'application de l'ensemble des conceptions des groupes dans l'ensemble des partitions de $[\![1; N]\!]$ qui à une conception associe la partition Γ définie ci-dessus est bijective.

Démonstration. Immédiat car chaque groupe de la partition correspond exactement au groupe de la conception par identification des individus à un entier de $[\![1; N]\!]$. □

De même, on parlera d'une partition de $[\![1; N]\!]$ pour représenter une unique manière de concevoir les groupes.

Deuxième partie
Population homogène

Dans cette partie, on suppose que la population est homogène. On ne raisonnera donc qu'avec des partitions τ de N.

3. Un premier critère de sélection : minimiser le nombre de tests

Pour choisir une conception d'un groupe, il nous faut un critère de sélection.

3.1. Nombre de tests

Définition 3.1

Pour une conception τ, on note $T(\tau)$ la variable aléatoire discrète qui associe le nombre de tests à l'issue du processus.

Remarque. La variable aléatoire T est discrète car finie. En effet, l'univers Ω correspond aux différents états de santé possibles de la population qui est de taille finie N. On peut même affirmer $|\Omega| = 2^N$.

Notre premier critère de sélection va ainsi être le suivant : on veut trouver τ minimisant $\mathbb{E}(T(\tau))$. On verra un autre critère dans la suite. On l'introduira le moment voulu.

3.2. Formule de l'espérance du nombre de test

Avant de déterminer les conceptions optimales, il faut connaître l'expression de l'espérance de la variable aléatoire ci-dessus.

Introduisons d'abord deux probabilités. Elles seront utiles dans le calcul de l'espérance.

Les tests d'une maladie n'étant pas parfaits, ils peuvent donner des faux positifs ou faux négatifs.

Définition 3.2

On appelle $Se \in [0;1]$ la *sensibilité*, c'est-à-dire la probabilité que le résultat d'un test positif soit vrai, *i.e.* la probabilité d'avoir un vrai positif.
On appelle $Sp \in [0;1]$ la *spécificité*, c'est-à-dire la probabilité que le résultat d'un test négatif soit vrai, *i.e.* la probabilité d'avoir un vrai négatif.

On admet la formule suivante (elle est issue de l'article de Aprahamian *et al.* [3]).

Propriété 3.1 : formule de l'espérance du nombre de test dans le cas homogène

Pour une conception des groupes τ, on a :

$$\boxed{\mathbb{E}(T(\tau)) = |\tau| + \sum_{\substack{n\in\tau \\ n\neq 1}} n(Se - (Se + Sp - 1)(1-p)^n).}$$

4. Test parfait

On va dans un premier temps supposer que le test est parfait. Cela revient à dire que $Se = Sp = 1$: il n'y a pas de faux positifs ou de faux négatifs.

4.1. Groupes de même taille

Cherchons tout d'abord à élaborer des groupes dont les tailles sont les mêmes. Notons n cette taille commune.

On exclut le cas $n = 1$ car il revient à tester individuellement chaque individu. De plus, $n > N$ est impossible à réaliser, d'où $n \in [\![2; N]\!]$.

On voit un premier problème : que faire si N n'est pas un multiple de n, *i.e.* si on ne pas former parfaitement des groupes de n sujets ?

On traitera ce cas plus tard, et on va d'abord supposer que $n|N$. On dira que ce cas est idéal.

4.1.1. Cas idéal

Soit n un diviseur de N. Alors $\exists\, q \in [\![1; N[\![$, $N = qn$. On crée donc q groupes de n sujets, et alors :

$$\tau = (\underbrace{n; n; \ldots; n}_{q \text{ termes}}).$$

L'entier q est unique et vaut $\frac{N}{n}$. Ainsi, à un entier n de $[\![2; N]\!]$, on associe une unique partition. En ce sens, on notera pour simplifier $T(n)$ au lieu de $T(\tau)$.

On cherche donc le diviseur n de N minimisant $\mathbb{E}(T(n))$ et tel que $n > 1$.

Remarque. Immédiatement, si N est premier, alors $n = N$ (car $n > 1$). C'est pourquoi on étudiera le cas plus tard non idéal où on considère n non diviseur de N.

Propriété 4.1

Soit n un diviseur de N tel que $n > 1$. Alors :

$$\boxed{\mathbb{E}(T(n)) = N\left(1 + \frac{1}{n} - (1-p)^n\right).}$$

Démonstration. On applique la formule 3.1 avec τ ci-dessus (qui ne contient aucun 1 car $n > 1$), et $Se = Sp = 1$:

$$\begin{aligned}
\mathbb{E}(T(n)) &= |\tau| + \sum_{a \in \tau} a(1 - (1-p)^a) \\
&= q + q \times n(1 - (1-p)^n) \\
&= q + qn - qn(1-p)^n \\
&= \frac{N}{n} + N - N(1-p)^n \\
\mathbb{E}(T(n)) &= N\left(1 + \frac{1}{n} - (1-p)^n\right).
\end{aligned}$$

□

Préliminaire : introduction à la fonction W de Lambert

Traçons le tableau de variation de la fonction $w : x \mapsto x\,\mathrm{e}^x$ définie et dérivable sur $\mathbb{R}$, de dérivée $w' : x \mapsto (x+1)\,\mathrm{e}^x$.

x	$-\infty$		-1		0		$+\infty$
$w'(x)$		$-$	0	$+$			
w	0	↘	$-\mathrm{e}^{-1}$	↗	0	↗	$+\infty$

On introduit dès lors la fonction W de Lambert, définie sur deux branches :
— pour $y \in [-\mathrm{e}^{-1}, 0[$, $W_{-1}(y)$ est l'unique solution dans $]-\infty, -1]$ de l'équation $y = x\,\mathrm{e}^x$;
— pour $y \in [-\mathrm{e}^{-1}, +\infty[$, $W_0(y)$ est l'unique solution dans $[-1, +\infty[$ de l'équation $y = x\,\mathrm{e}^x$.

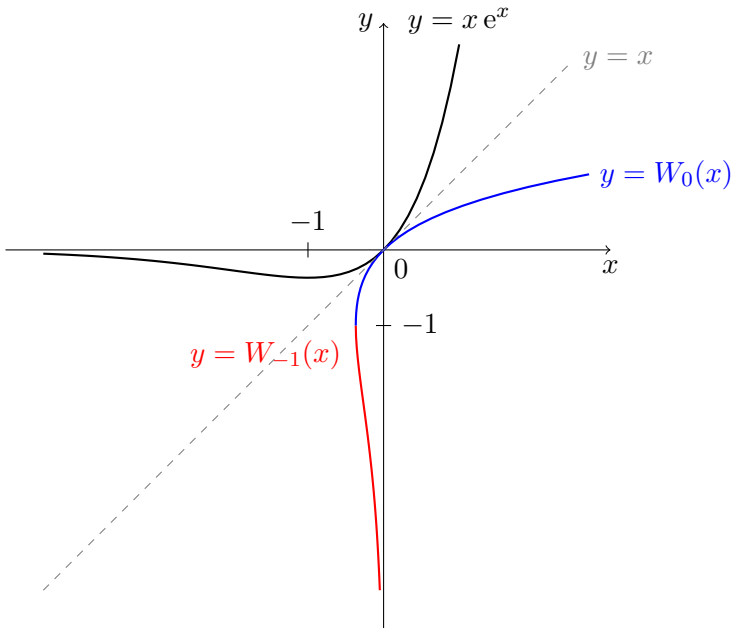


Figure 1 – Fonction W de Lambert

Théorème 4.1 : condition nécessaire d'optimalité

Soit n un diviseur de N, différent de 1. Pour que n soit optimal, *i.e.* donne une

espérance strictement plus petite que N, il est nécessaire que :

$$p < 1 - \mathrm{e}^{-\mathrm{e}^{-1}} \approx 0{,}307\,8 \qquad \text{et} \qquad n \in \left]\frac{W_0\big(\ln(1-p)\big)}{\ln(1-p)}; \frac{W_{-1}\big(\ln(1-p)\big)}{\ln(1-p)}\right[.$$

On posera dans la suite $\boxed{p_0 = 1 - \mathrm{e}^{-\mathrm{e}^{-1}}}$.

Démonstration. On a :

$$\begin{aligned}
\mathbb{E}\big(T(n)\big) < N &\iff 1 + \frac{1}{n} - (1-p)^n < 1 \\
&\iff \frac{1}{n} < (1-p)^n \\
&\iff 1 < n\,\mathrm{e}^{n\ln(1-p)} \\
&\iff n\ln(1-p)\,\mathrm{e}^{n\ln(1-p)} < \ln(1-p) \\
\mathbb{E}\big(T(n)\big) < N &\iff w\big(n\ln(1-p)\big) < \ln(1-p)
\end{aligned}$$

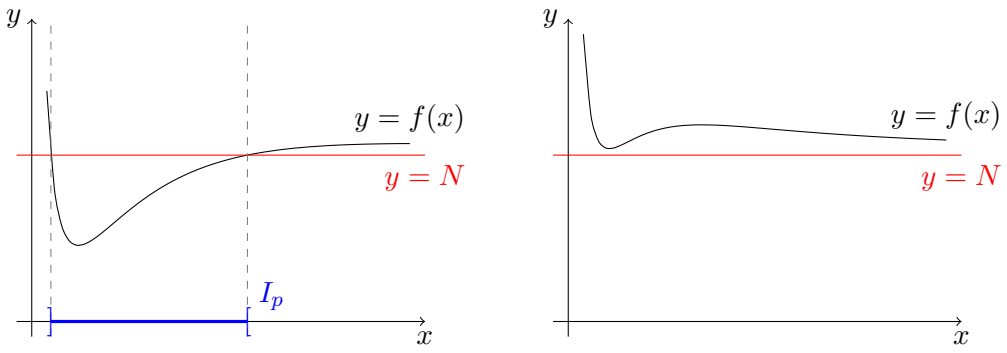
<u>Cas n° 1 :</u> $\ln(1-p) \leqslant -\mathrm{e}^{-1}$, c'est-à-dire $p \geqslant 1 - \mathrm{e}^{-\mathrm{e}^{-1}}$.
Alors comme le minimum de w est $-\mathrm{e}^{-1}$, il ne peut exister de n vérifiant l'inégalité.

<u>Cas n° 2 :</u> $\ln(1-p) > -\mathrm{e}^{-1}$, c'est-à-dire $p < 1 - \mathrm{e}^{-\mathrm{e}^{-1}}$.
On remarque que $\ln(1-p) < 0$. En particulier, l'étude de w faite ci-dessus donne :

$$\mathbb{E}\big(T(n)\big) < N \iff W_{-1}\big(\ln(1-p)\big) < n\ln(1-p) < W_0\big(\ln(1-p)\big)$$

d'où le résultat en divisant par $\ln(1-p) < 0$. □

Illustrons graphiquement ce résultat.



(a) cas $p < p_0$ (b) cas $p > p_0$

FIGURE 2 – Fonction $f : x \mapsto N\left(1 + \frac{1}{x} - (1-p)^x\right)$ selon la valeur de p

On étudiera plus tard plus en profondeur la fonction f.

Théorème 4.2 : condition suffisante d'optimalité

Si $N \geqslant 3$ et si $p \leqslant 1 - \mathrm{e}^{-\frac{\ln(N)}{N}}$, alors tout diviseur de N différent de 1 est optimal.

Démonstration. On suppose $N \geqslant 3$.
Plaçons nous dans le cas $p < p_0 = 1 - \mathrm{e}^{-\mathrm{e}^{-1}}$. D'après la démonstration précédente, un diviseur de N non égal à 1 est optimal si et seulement si il appartient à l'intervalle :

$$I_p = \left]\frac{W_0\big(\ln(1-p)\big)}{\ln(1-p)}; \frac{W_{-1}\big(\ln(1-p)\big)}{\ln(1-p)}\right[.$$

Recherchons les réels p dans $]0, p_0[$ tels que $\sup(I_p) \geqslant N$ et $\inf(I_p) \leqslant 2$.
Soient i et s les fonctions qui à p dans $]0; p_0[$ associent respectivement $\inf(I_p)$ et $\sup(I_p)$. Remarquons d'abord que par définition de W, on a :

$$i(p) = \exp\Big(-W_0\big(\ln(1-p)\big)\Big) \qquad \text{et} \qquad s(p) = \exp\Big(-W_{-1}\big(\ln(1-p)\big)\Big).$$

Ainsi, on a d'une part :

$$\begin{aligned}
i(p) \leqslant 2 &\iff \exp\Big(-W_0\big(\ln(1-p)\big)\Big) \leqslant 2 \\
&\iff -W_0\big(\ln(1-p)\big) \leqslant \ln(2) \\
&\iff W_0\big(\ln(1-p)\big) \geqslant -\ln(2) && \text{avec } -\ln(2) > -\ln(e) = -1 \\
&\iff \ln(1-p) \geqslant w(-\ln(2)) && \text{car } w \text{ croit sur }]-1; +\infty[\\
&\iff 1 - p \geqslant \exp\left(-\ln(2)\mathrm{e}^{-\ln(2)}\right) \\
i(p) \leqslant 2 &\iff p \leqslant 1 - \mathrm{e}^{-\frac{\ln(2)}{2}}.
\end{aligned}$$

Et d'autre part :

$$\begin{aligned}
s(p) \geqslant N &\iff \exp\Big(-W_{-1}\big(\ln(1-p)\big)\Big) \geqslant N \\
&\iff W_{-1}\big(\ln(1-p)\big) \leqslant -\ln(N) && \text{avec } -\ln(N) < -\ln(e) = -1 \\
&\iff \ln(1-p) \geqslant w\big(-\ln(N)\big) && \text{car } w \text{ décroit sur }]-\infty; -1[\\
s(p) \geqslant N &\iff p \leqslant 1 - \mathrm{e}^{-\frac{\ln(N)}{N}}.
\end{aligned}$$

La fonction $N \mapsto \frac{\ln(N)}{N}$ est de dérivée $N \mapsto \frac{1-\ln(N)}{N^2}$. En particulier, elle est décroissante sur $]\mathrm{e}; +\infty[$. Donc :

$$1 - \mathrm{e}^{-\frac{\ln(N)}{N}} < 1 - \mathrm{e}^{-\frac{\ln(\mathrm{e})}{\mathrm{e}}} = 1 - \mathrm{e}^{-\mathrm{e}^{-1}} = p_0 < 1 - \mathrm{e}^{-\frac{\ln(2)}{2}}$$

Dès lors, si $p \leqslant 1 - \mathrm{e}^{-\frac{\ln(N)}{N}}$, alors $p \in]0; p_0[$ et $p \leqslant 1 - \mathrm{e}^{-\frac{\ln(2)}{2}}$, et donc $i(p) \leqslant 2$ et $s(p) \geqslant N$.
En particulier, $[\![2; N]\!] \subset I_p$. Donc tout diviseur de N différent de 1 est optimal. □

Rappelons que l'on considère une population assez grande pour considérer les individus comme indépendants entre eux. En particulier, la condition $N \geqslant 3$ est superflue.

Posons $\boxed{h(N) = 1 - \mathrm{e}^{-\frac{\ln(N)}{N}}}$. Résumons les résultats ci-dessus :

— si $0 < p \leqslant h(N)$, il faut chercher parmi tous les diviseurs de N non égaux à 1 celui qui minimise l'espérance ;

— si $h(N) < p < p_0$, il faut chercher parmi tous les diviseurs de N non égaux à 1 et dans l'intervalle I_p celui qui minimise l'espérance, sous réserve d'existence ;

— si $p \geqslant p_0$, alors on ne peut trouver une conception des groupes de même taille qui rend la procédure efficace par rapport au cas « classique » où chaque individu est testé individuellement.

On voit alors que $h(N)$ est un seuil en dessous duquel l'existence d'une taille de groupe optimale est assurée. De même, p_0 est un seuil en dessus duquel il n'existe aucune conception optimale. Il est cependant difficile de conclure dans la zone $]h(N), p_0[$ car il se peut qu'aucun diviseur ne soit dans I_p. Donnons des exemples.

Exemple 4.1 : cas où il existe un diviseur optimal

Prenons $N = 100$. On calcule $h(N) \approx 0{,}045$. Prenons alors $p = 0{,}2 \in]h(N); p_0[$. On calcule d'abord l'intervalle I_p :

$$\frac{W_0(\ln(0{,}8))}{\ln(0{,}8)} \approx 1{,}35 \qquad \text{et} \qquad \frac{W_{-1}(\ln(0{,}8))}{\ln(0{,}8)} \approx 10{,}57.$$

On regarde ensuite d'après la formule 4.1 l'espérance pour les diviseurs de 100 (sauf pour 1) qui sont dans l'intervalle :

n	2	4	5	10	20	25	50	100
est dans l'intervalle I_p	oui	oui	oui	oui	non	non	non	non
$\mathbb{E}(T(n))$ (approximation)	86	84	87,2	99,3	103,8	103,6	102	101

Il existe ainsi dans ce cas une conception optimale qui consiste à ne former que des groupes de 4.

Exemple 4.2 : cas où il n'existe pas de diviseur optimal

Prenons encore $N = 100$. On rappelle $h(N) \approx 0{,}045$. Prenons cette fois-ci une probabilité $p = 0{,}3 \in]h(N); p_0[$. Les bornes de l'intervalle I_p deviennent :

$$\frac{W_0(\ln(0{,}7))}{\ln(0{,}7)} \approx 2{,}16 \qquad \text{et} \qquad \frac{W_{-1}(\ln(0{,}7))}{\ln(0{,}7)} \approx 3{,}56.$$

Comme 3 ne divise pas 100, d'après la condition nécessaire d'optimalité (4.1), il ne peut exister de conception avec des groupes de même taille optimale.

4.1.2. Cas quelconque : méthode de la division euclidienne

Étudions maintenant le cas où n n'est pas nécessairement un diviseur de N. On ne peut alors pas former exactement des groupes de taille n. Comment remédier à ce problème ?

Introduisons la division euclidienne de N par n :

$$\exists!\, q \in \mathbb{N},\ \exists!\, r \in [\![0; n-1]\!],\ N = nq + r.$$

On utilise cette unique décomposition pour créer la conception suivante : on forme q groupes de n individus et 1 groupe de r individu(s). *A priori*, r est non nul (on a déjà étudié le cas $r = 0$). Ainsi :

$$\tau = (\underbrace{n, \dots, n}_{q \text{ fois}}, r).$$

Les entiers q et r sont uniques à n. On peut donc pour simplifier noter $T(n)$ au lieu de $T(\tau)$, comme précédemment.

Remarque. On voit que la division euclidienne permet de déterminer deux partitions de N lorsque $q \neq n$ (en échangeant le rôle de n et q).

Propriété 4.2

Soit n un entier dans $[\![2; N]\!]$ qui ne divise pas N. On a :

— si $r = 1$, alors :

$$\boxed{\mathbb{E}(T(n)) = N + q - nq(1-p)^n};$$

— si $r > 1$, alors :

$$\boxed{\mathbb{E}(T(n)) = N + q + 1 - nq(1-p)^n - r(1-p)^r}.$$

Démonstration. Dans les deux cas, appliquons la formule 3.1 avec τ ci-dessus, et $Se = Sp = 1$:

— Supposons $r = 1$. Alors :

$$\begin{aligned}
\mathbb{E}(T(n)) &= |\tau| + \sum_{\substack{a \in \tau \\ a \neq 1}} a(1 - (1-p)^a) \\
&= q + 1 + nq(1 - (1-p)^n) \\
&= q + \underbrace{1 + nq}_{=N} - nq(1-p)^n \\
\mathbb{E}(T(n)) &= N + q - nq(1-p)^n.
\end{aligned}$$

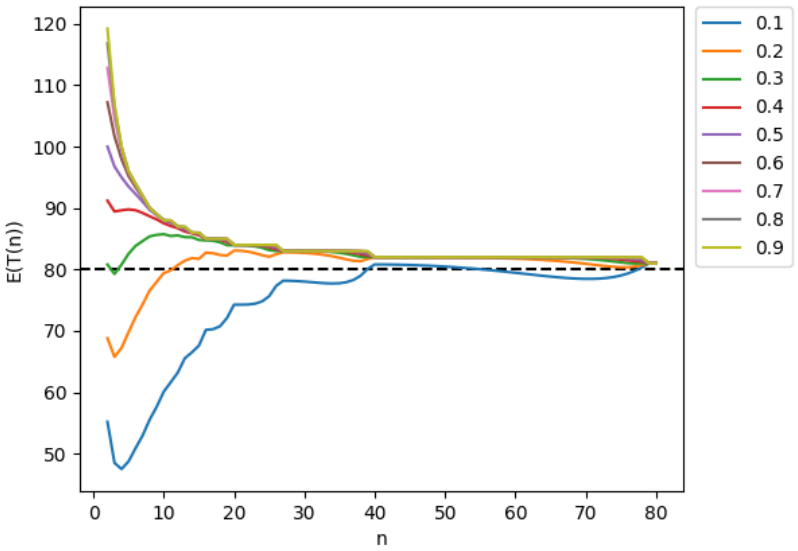


FIGURE 3 – Courbe de l'espérance de $T(n)$ en fonction de n pour $N = 80$

— Supposons $r > 1$. Alors :

$$\begin{aligned}\mathbb{E}(T(n)) &= |\tau| + \sum_{\substack{a\in\tau\\ a\neq 1}} a(1-(1-p)^a)\\ &= q+1+nq(1-(1-p)^n)+r(1-(1-p)^r)\\ &= q+1+\underbrace{nq+r}_{=N} -nq(1-p)^n - r(1-p)^r\\ \mathbb{E}(T(n)) &= N+q+1-nq(1-p)^n - r(1-p)^r. \end{aligned}$$ □

Remarque. Les valeurs de q et r dépendent de n. En fait, on pourrait voir q et r comme des fonctions de la variable n. En effet, on peut montrer que :

$$q = \left\lfloor \frac{N}{n} \right\rfloor \qquad \text{et} \qquad r = N - nq = N - n\left\lfloor \frac{N}{n} \right\rfloor.$$

En remplaçant q et r par leur expression explicite dans les formules ci-dessus, les expressions des espérances deviennent beaucoup plus complexes que dans le cas où n divise N. On ne va donc pas chercher à établir des résultats théoriques comme nous l'avions fait précédemment. On va plutôt établir des résultats graphiques, certes moins rigoureux mais beaucoup plus accessibles.

Regardons pour commencer la courbe de l'espérance de $T(n)$ en fonction de n, pour une population de taille N fixé.

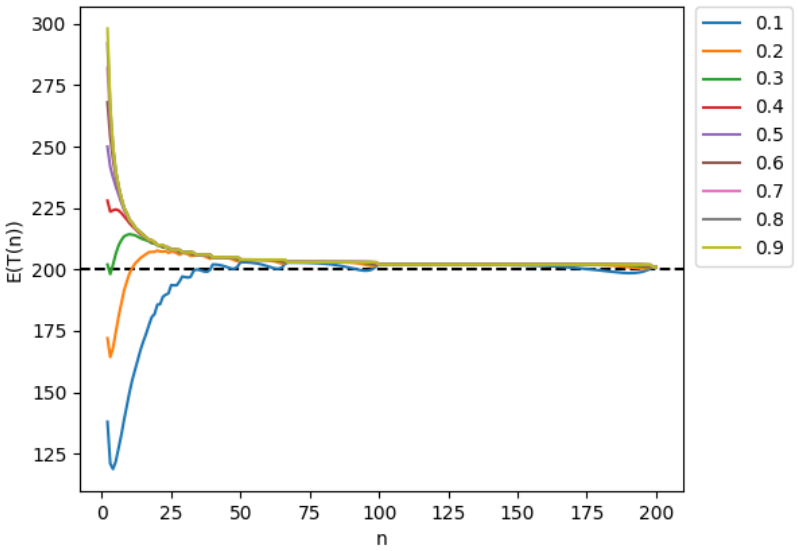


FIGURE 4 – Courbe de l'espérance de $T(n)$ en fonction de n pour $N = 200$

Code source 4.1 : programme de tracé des courbes 3 et 4, en Python

```
import matplotlib.pyplot as plt
import numpy as np


def esperanceT(N, n, p):
    if n == 1:
        return N
    q, r = N // n, N % n
    if r == 0:
        return N * (1 + 1/n - (1-p)**n)
    if r == 1:
        return N + q - n*q*(1-p)**n
    return N + q + 1 - n*q*(1-p)**n - r*(1-p)**r


def esperanceT_VS_n_ps(N):
    ps = np.arange(0.1, 1, 0.1)
    ns = np.arange(2, N+1) # on commence à 2 pour rendre les
    ↪  courbes plus claires
    fig = plt.figure("N = {}".format(N))
```

```
    ax = fig.add_axes([0.13, 0.15, 0.7, 0.75])
    ax.set_xlabel("n")
    ax.set_ylabel("E(T(n))")
    ax.axhline(N, linestyle="dashed", c="k")
    for p in ps:
        es = [esperanceT(N, n, p) for n in ns]
        ax.plot(ns, es, label = str(round(p,1)))
    ax.legend(bbox_to_anchor=(1.02, 1), loc='upper left',
    ↪  borderaxespad=0.)


esperanceT_VS_n_ps(80)
esperanceT_VS_n_ps(200)
plt.show()
```

On observe sur les courbes 3 et 4 qu'il semble aussi exister un seuil de probabilité au delà duquel aucune valeur de n ne soit optimale.

Calculons alors numériquement ce seuil, à l'aide du programme ci-contre.

Code source 4.2 : détermination du seuil d'optimalité pour la méthode de la division euclidienne, en Python

```
def esperanceT(N, n, p):
    if n == 1:
        return N
    q, r = N // n, N % n
    if r == 0:
        return N * (1 + 1/n - (1-p)**n)
    if r == 1:
        return N + q - n*q*(1-p)**n
    return N + q + 1 - n*q*(1-p)**n - r*(1-p)**r


def espT_minimale(N, p):
    min_e = N
    for n in range(2, N+1):
        e = esperanceT(N, n, p)
        if e < min_e:
            min_e = e
    return min_e
```

```python
def seuil(N, p0, nbDecimales):
    p = p0
    e = espT_minimale(N, p)
    while e < N:
        p += 10 ** (- nbDecimales)
        e = espT_minimale(N, p)
    return p - 10 ** (- nbDecimales)


def calcul_seuil(N, nbDecimales):
    p = 0
    for k in range(1, nbDecimales + 1):
        p = seuil(N, p, k)
    return p


for N in [10, 20, 50, 80, 100, 101, 200, 500, 1000, 2000, 5000,
↪   10000]:
    print(N, round(calcul_seuil(N, 8),8))
```

On obtient les résultats suivants :

N	seuil
10	0,306 638 72
20	0,305 301 29
50	0,306 104 41
80	0,306 304 88
100	0,306 638 72
101	0,306 374 32
200	0,306 505 22
500	0,306 585 33
1 000	0,306 638 72
2 000	0,306 625 37
5 000	0,306 633 38
10 000	0,306 638 72

Les seuils semblent spécifiques à N mais sont très proches (même pour N premier, *cf.* $N = 101$). Remarquons aussi leur proximité avec $p_0 = 1 - \mathrm{e}^{-\mathrm{e}^{-1}} \approx 0{,}307\,799\,37$. On conjecture alors un lien entre le cas idéal et le cas quelconque.

Pour mettre en évidence ce lien, comparons pour tout n dans $[\![2; N]\!]$ l'espérance de $T(n)$ avec $f(n) = N\left(1 + \frac{1}{n} - (1-p)^n\right)$ (espérance de $T(n)$ dans le cas idéal, *cf.* 4.1).

On va par la suite seulement considérer des probabilités en dessous des seuils obtenus ci-dessus (on ne s'intéresse pas aux cas où aucune valeur de n n'est optimale).

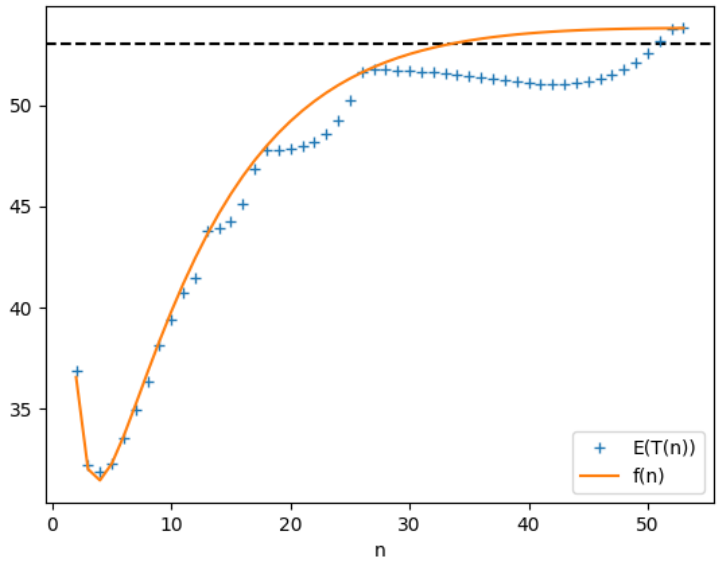


FIGURE 5 – Comparaison de $\mathbb{E}(T(n))$ et de $f(n)$ pour $N = 53$ et $p = 0{,}1$

Considérons le cas extrême : on prend N premier de telle sorte que l'écart entre le cas quelconque et le cas idéal soit le plus important.

Code source 4.3 : programme de tracé des courbes 5 et 6, en Python

```
import matplotlib.pyplot as plt


def esperanceT(N, n, p):
    q, r = N // n, N % n
    if r == 0:
        return N * (1 + 1/n - (1-p)**n)
    if r == 1:
        return N + q - n*q*(1-p)**n
    return N + q + 1 - n*q*(1-p)**n - r*(1-p)**r


def f(N, n, p):
    return N * (1 + 1/n - (1-p)**n)


def superposer(N, p):
```

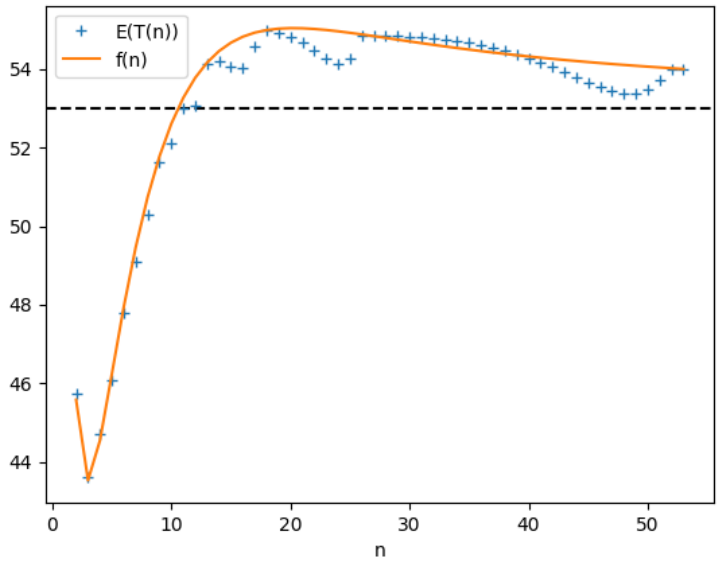


FIGURE 6 – Comparaison de $\mathbb{E}(T(n))$ et de $f(n)$ pour $N = 53$ et $p = 0{,}2$

```
    ns = range(2, N+1)
    espT = [esperanceT(N, n, p) for n in ns]
    ideal = [f(N, n, p) for n in ns]
    plt.figure("N = {} ; p = {}".format(N,p))
    plt.axhline(N, linestyle = "dashed", color = "k")
    plt.plot(ns, espT, "+", label = "E(T(n))")
    plt.plot(ns, ideal, label = "f(n)")
    plt.xlabel("n")
    plt.legend()


superposer(53, 0.1)
superposer(53, 0.2)
plt.show()
```

On observe que dans les deux cas, les courbes coïncident bien pour les faibles valeurs de n tandis qu'elles diffèrent pour les grandes valeurs de n. Le plus important est de remarquer que les minima coïncident bien.

On conjecture alors le résultat suivant : le minimum de $\mathbb{E}(T(n))$ et celui de $f(n)$ sont atteints au même point.

Encore une fois, regardons numériquement si le résultat est vrai.

On crée une fonction qui pour un entier N donné calcule pour une série de valeurs de p comprises entre 0 et p_0 (bonne approximation du seuil pour N) l'écart entre le minimum de f et celui de l'espérance de $T(n)$. On présente les résultats sous la forme d'un tableau de couple de la forme (e, nb) où e est l'écart et nb le nombre de probabilités ayant donné un tel écart.

Code source 4.4 : tableau des fréquences des écarts entre le minimum de f et de l'espérance de $T(n)$, en Python

```python
import numpy as np


p0 = 1 - np.exp(-np.exp(-1))


def esperanceT(n, N, p):
    if n == 1:
        return N
    q, r = N // n, N % n
    if r == 0:
        return N * (1 + 1/n - (1-p)**n)
    if r == 1:
        return N + q - n*q*(1-p)**n
    return N + q + 1 - n*q*(1-p)**n - r*(1-p)**r


def fIdeal(n, N, p):
    if n == 1:
        return N
    return N * (1 + 1/n - (1-p)**n)


def min_fonction(f, N, p):
    min_n, min_e = 1, f(1, N, p)
    for n in range(2, N+1):
        e = f(n, N, p)
        if e < min_e:
            min_n, min_e = n, e
    return min_n


def nbEcart_NFixe(N):
    ps = np.linspace(0.001, p0, 1000)
```

```
    ecarts = {}
    for p in ps:
        min_n_fIdeal = min_fonction(fIdeal, N, p)
        min_n_espT = min_fonction(esperanceT, N, p)
        e = abs(min_n_fIdeal - min_n_espT)
        try:
            ecarts[e] += 1
        except KeyError:
            ecarts[e] = 1
    return sorted(ecarts.items())


for N in [10, 20, 53, 80, 100, 200, 500, 1000, 2000]:
    print(N, nbEcart_NFixe(N))
```

On obtient les résultats suivants :

```
10 [(0, 592), (1, 368), (2, 31), (3, 9)]
20 [(0, 845), (1, 128), (2, 16), (3, 4), (4, 4), (5, 2), (6, 1)]
53 [(0, 840), (1, 148), (2, 8), (3, 1), (4, 1), (5, 2)]
80 [(0, 906), (1, 85), (2, 7), (4, 1), (5, 1)]
100 [(0, 911), (1, 83), (2, 5), (3, 1)]
200 [(0, 937), (1, 53), (2, 10)]
500 [(0, 974), (1, 23), (2, 3)]
1000 [(0, 975), (1, 24), (3, 1)]
2000 [(0, 988), (1, 12)]
```

Plus N est grand, plus les minima sont proches. Comme on considère N grand, il est alors légitime d'approximer l'entier n minimisant l'espérance dans le cas quelconque par celui de f. Continuons alors l'étude de f sur $\mathbb{R}_+^*$, déjà entamée dans le cas idéal, pour trouver la valeur qui minimise la fonction.

Préliminaire : introduction à une autre fonction implicite

Soit z la fonction définie sur $\mathbb{R}$ qui à x associe $x^2\,\mathrm{e}^x$. Elle est dérivable sur $\mathbb{R}$ de dérivée $x \mapsto x(x+2)\mathrm{e}^x$.

x	$-\infty$		-2		0		$+\infty$
x		$-$		$-$	0	$+$	
$x+2$		$-$	0	$+$		$+$	
z	0	↗	$4\mathrm{e}^{-2}$	↘	0	↗	$+\infty$

De la même manière que pour la fonction W, on introduit la fonction Z définie sur 3 branches :

— pour $y \in]0; 4\mathrm{e}^{-2}]$, $Z_-(y)$ est l'unique solution dans $]-\infty, -2]$ de l'équation $y = x^2\,\mathrm{e}^x$ d'inconnue x ;

— pour $y \in [0; 4\mathrm{e}^{-2}]$, $Z_0(y)$ est l'unique solution dans $[-2, 0]$ de l'équation $y = x^2\,\mathrm{e}^x$;

— pour $y \in [0; +\infty[$, $Z_+(y)$ est l'unique solution dans $[0; +\infty[$ de l'équation $y = x^2\,\mathrm{e}^x$.

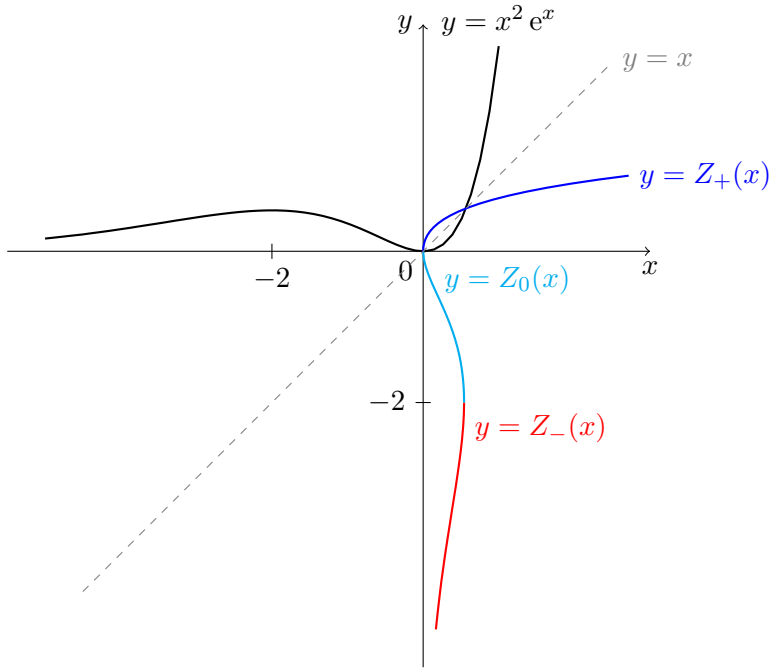


FIGURE 7 – Fonction Z

Calculons maintenant la dérivée de f, définie sur $\mathbb{R}_+^*$:

$$\forall x > 0,\ f'(x) = N\left(-\frac{1}{x^2} - \ln(1-p)\cdot(1-p)^x\right)$$

On a donc pour $x > 0$:

$$\begin{aligned}
f'(x) \geqslant 0 &\iff \frac{1}{x^2} + \ln(1-p)\cdot(1-p)^x \leqslant 0\\
&\iff 1 + \ln(1-p)\,x^2\,\mathrm{e}^{\ln(1-p)x} \leqslant 0\\
&\iff \ln(1-p)\,x^2\,\mathrm{e}^{\ln(1-p)x} \leqslant -1\\
&\iff \ln^2(1-p)\,x^2\,\mathrm{e}^{\ln(1-p)x} \geqslant -\ln(1-p)\\
f'(x) \geqslant 0 &\iff z\big(\ln(1-p)\,x\big) \geqslant -\ln(1-p).
\end{aligned}$$

L'équation $z(x) = y$ d'inconnue x peut avoir de 0 à 3 solutions selon la valeur de y d'après ce qui précède. Faisons alors une disjonction de cas sur $-\ln(1-p)$. Remarquons au préalable que $-\ln(1-p) > 0$ et donc qu'il y a au moins une solution.

On a $-\ln(1-p) \geqslant 4\mathrm{e}^{-2} \iff 1-p \leqslant \mathrm{e}^{-4\mathrm{e}^{-2}} \iff p \geqslant 1-\mathrm{e}^{-4\mathrm{e}^{-2}}$.

Posons $\boxed{p_1 = 1-\mathrm{e}^{-4\mathrm{e}^{-2}}}$. On a $p_1 \approx 0{,}418\,03 > p_0$. On ne s'intéresse donc pas au cas $-\ln(1-p) \geqslant 4\mathrm{e}^{-2}$. On va donc supposer que $-\ln(1-p) < 4\mathrm{e}^{-2}$, *i.e.* $p < p_1$. C'est le cas lorsque $p \leqslant p_0$.

Alors d'après l'étude de la fonction z ci-dessus, on a :

$$\begin{aligned}
f'(x) \geqslant 0 &\iff \ln(1-p)\,x \in \big[Z_-\big(-\ln(1-p)\big); Z_0\big(-\ln(1-p)\big)\big]\\
&\qquad \text{ou} \qquad \ln(1-p)\,x \geqslant Z_+\big(-\ln(1-p)\big)\\
&\iff x \in \left[\frac{Z_0\big(-\ln(1-p)\big)}{\ln(1-p)}; \frac{Z_-\big(-\ln(1-p)\big)}{\ln(1-p)}\right]\\
&\qquad \text{ou} \qquad x \leqslant \underbrace{\frac{Z_+\big(-\ln(1-p)\big)}{\ln(1-p)}}_{<0}\\
f'(x) \geqslant 0 &\iff x \in \left[\frac{Z_0\big(-\ln(1-p)\big)}{\ln(1-p)}; \frac{Z_-\big(-\ln(1-p)\big)}{\ln(1-p)}\right]
\end{aligned}$$

Posons :

$$\boxed{n_0(p) = \frac{Z_0\big(-\ln(1-p)\big)}{\ln(1-p)}} \qquad \text{et} \qquad \boxed{n_-(p) = \frac{Z_-\big(-\ln(1-p)\big)}{\ln(1-p)}}.$$

On en déduit les variations de f sur $\mathbb{R}_+^*$:

x	0		$n_0(p)$		$n_-(p)$		$+\infty$
$f'(x)$	‖	$-$	0	$+$	0	$-$	
f	‖ $+\infty$	↘	$f(n_0(p))$	↗	$f(n_-(p))$	↘	N

Il nous reste à étudier le point $n_0(p)$ où l'espérance est minimale.

D'après le théorème de la dérivée d'une bijection réciproque, Z_0 est dérivable sur $]0;4\mathrm{e}^{-2}[$, et pour x dans $]0;4\mathrm{e}^{-2}[$:

$$\begin{aligned} Z_0'(x) &= \frac{1}{z'(Z_0(x))} \\ &= \frac{1}{Z_0(x)(Z_0(x)+2)\,\mathrm{e}^{Z_0(x)}} \\ Z_0'(x) &= \frac{Z_0(x)}{x(Z_0(x)+2)} \end{aligned}$$

car $Z_0(x)\mathrm{e}^{Z_0(x)} = \dfrac{x}{Z_0(x)}$.

Donc n_0 est dérivable sur $]0;p_1[$, et pour $p \in]0;p_1[$:

$$\begin{aligned} n_0'(p) &= \frac{1}{\ln^2(1-p)}\left(\frac{\mathrm{d}Z_0\big(-\ln(1-p)\big)}{\mathrm{d}p}\ln(1-p) - Z_0\big(-\ln(1-p)\big)\frac{\mathrm{d}\ln(1-p)}{\mathrm{d}p}\right) \\ &= \frac{1}{\ln^2(1-p)}\left(\frac{1}{1-p}\cdot\frac{Z_0\big(-\ln(1-p)\big)}{-\ln(1-p)\cdot\big(Z_0\big(-\ln(1-p)\big)+2\big)}\ln(1-p) + \frac{Z_0\big(-\ln(1-p)\big)}{1-p}\right) \\ n_0'(p) &= \underbrace{\frac{Z_0\big(-\ln(1-p)\big)}{\ln^2(1-p)\cdot(1-p)}}_{<0}\left(1 - \frac{1}{Z_0\big(-\ln(1-p)\big)+2}\right). \end{aligned}$$

Ainsi, comme $\forall x \in]0;4\mathrm{e}^{-2}[$, $Z_0(x) \geqslant -2$ donc $Z_0(x)+2 \geqslant 0$, on a :

$$\begin{aligned} n_0'(p) \geqslant 0 &\iff 1 - \frac{1}{Z_0\big(-\ln(1-p)\big)+2} \leqslant 0 \\ &\iff 1 \leqslant \frac{1}{Z_0\big(-\ln(1-p)\big)+2} \\ &\iff Z_0\big(-\ln(1-p)\big)+2 \leqslant 1 \\ &\iff Z_0\big(-\ln(1-p)\big) \leqslant -1 \\ &\iff -\ln(1-p) \geqslant z(-1) \\ &\iff \ln(1-p) \leqslant -\mathrm{e}^{-1} \\ &\iff 1-p \leqslant \mathrm{e}^{-\mathrm{e}^{-1}} \\ n_0'(p) \geqslant 0 &\iff p \geqslant p_0. \end{aligned}$$

On a au voisinage de 0 :

$$n_0(p) = \frac{Z_0\big(-\ln(1-p)\big)}{-\left({Z_0}^2\big(-\ln(1-p)\big)\,\mathrm{e}^{Z_0\left(-\ln(1-p)\right)}\right)} = \frac{1}{-Z_0\big(-\ln(1-p)\big)\mathrm{e}^{Z_0\left(-\ln(1-p)\right)}}.$$

Or, d'après le théorème de bijection, Z_0 est continue en 0, donc :

$$Z_0\big(-\ln(1-p)\big) \xrightarrow[p\to 0]{} Z_0(0) = 0^-,$$

d'où $n_0(p) \xrightarrow[p\to 0]{} +\infty$.

De plus, on a :

$$n_0(p_0) = \frac{Z_0\big(-\ln(1-p_0)\big)}{\ln(1-p_0)} = \frac{Z_0\big(e^{-1}\big)}{-\mathrm{e}^{-1}} = \frac{-1}{-\mathrm{e}^{-1}} = \mathrm{e}$$

car $z(-1) = (-1)^2\mathrm{e}^{-1} = \mathrm{e}^{-1}$.

En bonus, on a :

$$n_0(p_1) = \frac{Z_0\big(4\mathrm{e}^{-2}\big)}{-4\mathrm{e}^{-2}} = \frac{-2}{-4\mathrm{e}^{-2}} = \frac{\mathrm{e}^2}{2}.$$

D'où le tableau de variation de n_0 :

p	0		p_0		p_1
$n_0'(p)$	‖	$-$	0	$+$	‖
n_0	‖ $+\infty$	↘	e	↗	$\frac{e^2}{2}$

Une conséquence importante est que $n_0(p) > 2$ pour p dans $]0; p_1]$, et donc pour $p < p_0$ (on rappelle qu'on ne s'intéresse qu'aux peobabilités p tels que $p < p_0$).

Il nous reste à chercher p tel que $n_0(p) \leqslant N$:

$$n_0(p) \leqslant N \iff \frac{Z_0\big(-\ln(1-p)\big)}{\ln(1-p)} \leqslant N \iff Z_0\big(-\ln(1-p)\big) \geqslant N\ln(1-p).$$

Or, on a :

$$N\ln(1-p) < -2 \iff 1-p < \exp\left(-\frac{2}{N}\right) \iff p > 1-\exp\left(-\frac{2}{N}\right).$$

On veut maintenant savoir la position de p_0 par rapport à $s(N)$, où $\boxed{s(N) = 1-\mathrm{e}^{-\frac{2}{N}}}$. On a :

$$\begin{aligned} p_0 \leqslant s(N) &\iff 1-\mathrm{e}^{-\mathrm{e}^{-1}} \leqslant 1-\mathrm{e}^{-\frac{2}{N}} \\ &\iff \mathrm{e}^{-1} \leqslant \frac{2}{N} \\ p_0 \leqslant s(N) &\iff N \leqslant 2\mathrm{e}. \end{aligned}$$

Comme la population est grande, on peut supposer $N \geqslant 2\mathrm{e} \approx 5{,}4$ (ou encore $N \geqslant 6$). Dans ce cas, le seuil $s(N)$ est en-dessous de p_0 : il est important de distinguer selon ce que p est en dessous ou en dessus de ce seuil.

Si $p > s(N)$, on a nécessairement $Z_0(-\ln(1-p)) \geqslant N\ln(1-p)$ car $N\ln(1-p) < -2$, donc $n_0(p) \leqslant N$.

Supposons alors $p \leqslant s(N)$.

$$\begin{aligned}
n_0(p) \leqslant N &\iff -\ln(1-p) \leqslant z\big(N\ln(1-p)\big)\\
&\iff -\ln(1-p) \leqslant N^2\ln^2(1-p)\,\mathrm{e}^{N\ln(1-p)}\\
&\iff N^2\ln(1-p)\,\mathrm{e}^{N\ln(1-p)} \leqslant -1\\
&\iff N\ln(1-p)\,\mathrm{e}^{N\ln(1-p)} \leqslant -\frac{1}{N}\\
n_0(p) \leqslant N &\iff w\big(N\ln(1-p)\big) \leqslant -\frac{1}{N}
\end{aligned}$$

Or, on a :

$$-\frac{1}{N} < -\mathrm{e}^{-1} \iff \frac{1}{N} > \mathrm{e}^{-1} \iff N < \mathrm{e}.$$

Ce cas n'est pas possible car on a supposé $N \geqslant 6$.

Ainsi :

$$\begin{aligned}
n_0(p) \leqslant N &\iff W_{-1}\left(-\frac{1}{N}\right) \leqslant N\ln(1-p) \leqslant W_0\left(-\frac{1}{N}\right)\\
&\iff \exp\left(\frac{1}{N}W_{-1}\left(-\frac{1}{N}\right)\right) \leqslant 1-p \leqslant \exp\left(\frac{1}{N}W_0\left(-\frac{1}{N}\right)\right)\\
n_0(p) \leqslant N &\iff 1-\exp\left(\frac{1}{N}W_0\left(-\frac{1}{N}\right)\right) \leqslant p \leqslant 1-\exp\left(\frac{1}{N}W_{-1}\left(-\frac{1}{N}\right)\right).
\end{aligned}$$

En tout, on a trouvé 3 seuils, que l'on doit comparer. Il s'agit de :

$$s(N) = 1-\mathrm{e}^{-\frac{2}{N}}; \quad \boxed{s_{-1}(N) = 1-\mathrm{e}^{\frac{1}{N}W_{-1}\left(-\frac{1}{N}\right)}} \quad \text{et} \quad \boxed{s_0(N) = 1-\mathrm{e}^{\frac{1}{N}W_0\left(-\frac{1}{N}\right)}}.$$

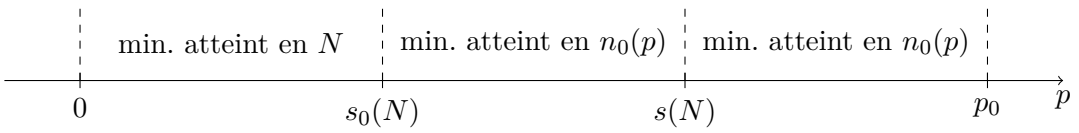
D'une part, $W_0\left(-\frac{1}{N}\right) \geqslant -1 > -2$ donc $\frac{1}{N}W_0\left(-\frac{1}{N}\right) > -\frac{2}{N}$ et donc $s_0(N) < s(N)$.

D'autre part, on a $N > \frac{\mathrm{e}^2}{2} \approx 3{,}7$ donc $-\frac{1}{N} > -2\mathrm{e}^{-2} = w(-2)$ donc $W_{-1}\left(-\frac{1}{N}\right) < -2$ par stricte décroissance. On en déduit comme ci-dessus que $s(N) < s_{-1}(N)$.

Ainsi, le seuil $s_{-1}(N)$ est trop grand par rapport à $s(N)$, au contraire du seuil $s_0(N)$. Donc :

$$n_0(p) \leqslant N \iff s_0(N) = 1-\mathrm{e}^{\frac{1}{N}W_0\left(-\frac{1}{N}\right)} \leqslant p \leqslant s(N) = 1-\mathrm{e}^{-\frac{2}{N}}.$$

Lorsque $n_0(p) > N$, il faut choisir $n = N$ car la fonction f est alors décroissante sur $[2; N]$. Résumons les résultats obtenus (avec toujours $N \geqslant 6$) :



On voit qu'en réalité, le seuil $s(N)$ n'a pas d'importance dans la détermination du minimum. Tout ce qui compte est le seuil $s_0(N)$.

Remarque. Toute cette étude a montré l'existence du seuil $s_0(N)$ passé inapperçu dans les études graphiques ci-dessus. En réalité, s_0 décroît tellement vite qu'il est difficile d'observer le minimum aller au delà de N : il faut choisir des valeurs de p très faibles :

N	6	10	100	1 000
$s_0(N)$ (approximation)	0,033 506	0,011 121	0,000 101	0,000 001

On résume tous ces résultats sous la forme du théorème ci-dessous :

Théorème 4.3

Si $N \geqslant 6$, en posant $s_0(N) = 1 - \mathrm{e}^{\frac{1}{N} W_0\left(-\frac{1}{N}\right)}$, on a :
- si $p \leqslant s_0(N)$, alors N minimise $\mathbb{E}\big(T(n)\big)$;
- si $s_0(N) < p < p_0$, alors l'entier le plus proche de $\dfrac{Z_0\big(-\ln(1-p)\big)}{\ln(1-p)}$ est une bonne approximation de l'entier n dans $[\![2; N]\!]$ minimisant $\mathbb{E}\big(T(n)\big)$.

Remarque. On a aussi une approximation de l'espérance minimale. Il s'agit de :

$$\begin{aligned} f\big(n_0(p)\big) &= N\left(1 + \frac{1}{n_0(p)} - \mathrm{e}^{\ln(1-p)\, n_0(p)}\right) \\ &= N\left(1 + \frac{\ln(1-p)}{Z_0\big(-\ln(1-p)\big)} - \mathrm{e}^{Z_0\left(-\ln(1-p)\right)}\right) \\ f\big(n_0(p)\big) &= N\left(1 + \frac{\ln(1-p)}{Z_0\big(-\ln(1-p)\big)} + \frac{\ln(1-p)}{Z_0^2\big(-\ln(1-p)\big)}\right) \end{aligned}$$

À l'aide de la première égalité, on remarque que $p \mapsto f\big(n_0(p)\big)$ est une fonction décroissante : au plus faible la probabilité p est, au mieux la procédure sera optimale. C'est ce qu'on avait conjecturé en introduction.

Remarque. Ce résultat complète le théorème 4.2. On peut même montrer que pour $N > \mathrm{e}$, $s_0(N) < h(N)$.

Remarque. On a vu que les seuils d'optimalité n'étaient avec exacetement p_0 mais plus aux alentours de $p_0 - 0{,}001$. Cela pourrait poser problème si $p_0 - 0{,}001 < s_0(N) < p_0$. En fait, cela ne sera jamais le cas car on peut montrer que s_0 est une fonction décroissante et que $s_0(6) \ll p_0 - 0{,}001$. On pourrait alors remplacer dans le théorème ci-dessus p_0 par la valeur $\widetilde{p}_0(N) \approx p_0 - 0{,}001$ définie comme étant le seuil d'optimalité propre à N.

Remarque. La courbe représentative de la fonction $p \mapsto n_0(p) = \dfrac{Z_0\big(-\ln(1-p)\big)}{\ln(1-p)}$ est présentée figure 8. On remarque que c'est une fonction décroissante qui croît très vite au voisinage de 0.

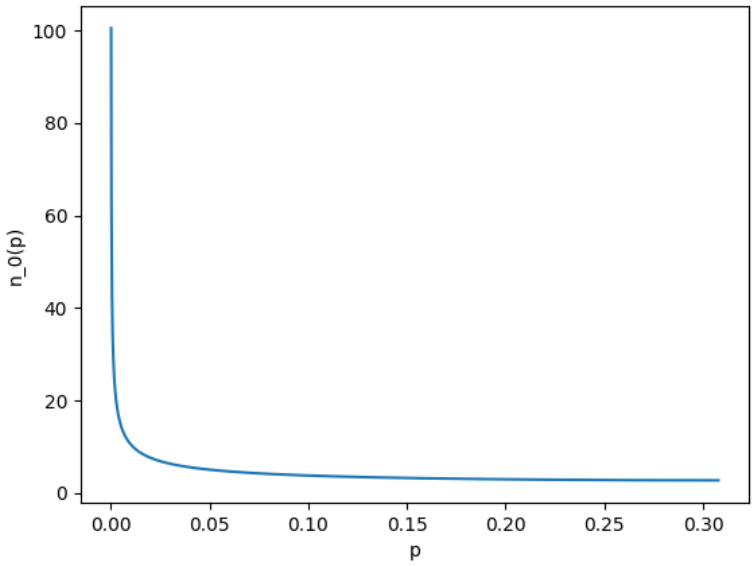


FIGURE 8 – Fonction $p \mapsto n_0(p)$ sur $]0; p_0[$

4.1.3. Une décomposition plus fidèle

La décomposition à l'aide de la division euclidienne peut parfois donner des groupes de grandes tailles (n) et un groupe d'une petite taille (r). Cela va à l'encontre du but même de cette section qui est de former des groupes de même taille. Par exemple, la décomposition $79 = 13 \times 6 + 1$ donne 6 groupes de 13 et un tout petit groupe de 1.

On propose alors une autre décomposition, plus fidèle à la philosophie de cette section.

Rappelons que le problème majeur réside dans le fait qu'il est impossible de créer des groupes de même taille si n ne divise pas N. La division euclidienne nous a permis un prolongement, mais pas parfait. En fait, on peut partir de cette décomposition pour en trouver une meilleure. Reprenons $79 = 13 \times 6 + 1$. On va prendre dans tous les groupes de 13 un individu que l'on vient ajouter dans le groupe solitaire : $79 = 12 \times 6 + 6 + 1 = 12 \times 6 + 7$. On se retrouve avec 6 groupes de 12 et un groupe de 7. On peut encore faire mieux en reprenant un individu à 4 groupes de 12 : $79 = 12 \times 2 + 11 \times 4 + 4 + 7 = 12 \times 2 + 11 \times 5$. D'où une décomposition finale de 2 groupes de 12 et 5 groupes de 11 : c'est beaucoup mieux que la division euclidienne ; on a crée des groupes dont la taille ne diffère que de 1.

En prenant successivement des individus dans des « gros » groupes pour aggrandir le petit groupe, on arrive à créer une décomposition plus équilibrée avec des groupes dont les tailles sont deux entiers consécutifs. Cette décomposition existera toujours dans le cas général ; elle provient du théorème suivant :

Théorème 4.4 : décomposition en combinaison linéaire de deux entiers consécutifs

Pour tout $n, q \in \mathbb{N}^*$ et pour $r \in [\![0; n-1]\!]$, en posant $N = nq + r$, il existe $\alpha \in [\![0; N-1]\!]$, $u \in [\![0; q]\!]$ et $v \in [\![1; q+1]\!]$ tels que :

$$N = \alpha u + (\alpha + 1)v.$$

Démonstration. Posons :

$$\begin{aligned}
\alpha &= n - 1 + \left\lfloor \frac{q + r - n}{q+1} \right\rfloor ; \\
u &= n - r + (q+1) \left\lfloor \frac{q + r - n}{q+1} \right\rfloor ; \\
v &= r - n + (q+1) \left(1 - \left\lfloor \frac{q + r - n}{q+1} \right\rfloor \right).
\end{aligned}$$

Calculons :

$$u + v = (q+1) \left(\left\lfloor \frac{q + r - n}{q+1} \right\rfloor + 1 - \left\lfloor \frac{q + r - n}{q+1} \right\rfloor \right) = q + 1.$$

On a donc :

$$\begin{aligned}
\alpha u + (\alpha + 1)v &= \alpha(u + v) + v \\
&= (q+1) \left(n - 1 + \left\lfloor \frac{q + r - n}{q+1} \right\rfloor \right) \\
&\qquad + r - n + (q+1) \left(1 - \left\lfloor \frac{q + r - n}{q+1} \right\rfloor \right) \\
&= (q+1)n + r - n \\
&= qn + n + r - n \\
\alpha u + (\alpha + 1)v &= nq + r.
\end{aligned}$$

Montrons maintenant que $\alpha \in \mathbb{N}$. C'est clairement un entier. En outre, écrivons :

$$\begin{aligned}
\alpha &= \left\lfloor n - 1 + \frac{q + r - n}{q+1} \right\rfloor \\
&= \left\lfloor \frac{nq + n - q - 1 + q + r - n}{q+1} \right\rfloor \\
&= \left\lfloor \frac{nq + r - 1}{q+1} \right\rfloor \\
\alpha &= \left\lfloor \frac{N-1}{q+1} \right\rfloor
\end{aligned}$$

Or, n et q sont non nuls donc $N \geqslant 1$ donc $\frac{N-1}{q+1} \geqslant 0$ donc $\alpha \geqslant 0$. De plus, $\alpha \leqslant \frac{N-1}{q+1} \leqslant N - 1$.

Étudions ensuite u et v. Ce sont clairement des entiers. Déterminons leur signe. On a :

$$\begin{aligned} u \geqslant 0 &\iff n - r \geqslant (q+1)\left\lfloor \frac{q+r-n}{q+1} \right\rfloor \\ &\iff \left\lfloor \frac{q+r-n}{q+1} \right\rfloor \leqslant \frac{n-r}{q+1} \\ u \geqslant 0 &\iff \left\lfloor \frac{q}{q+1} - \frac{r-n}{q+1} \right\rfloor \leqslant \frac{n-r}{q+1}. \end{aligned}$$

Soit $d \in [0;1[$. La fonction $x \mapsto \lfloor d - x \rfloor$ est décroissante sur $\mathbb{R}$ car la fonction partie entière est croissante. Donc $\forall x \geqslant 0$, $\lfloor d - x \rfloor \leqslant \lfloor d - 0 \rfloor = \lfloor d \rfloor = 0$ donc $\lfloor d - x \rfloor \leqslant x$. Or, comme $q > 0$, on a $\frac{q}{q+1} \in [0;1[$. Et $n - r > 0$ donc $\frac{n-r}{q+1} \geqslant 0$. D'après ce qui précède, on a alors $\left\lfloor \frac{q}{q+1} - \frac{r-n}{q+1} \right\rfloor \leqslant \frac{n-r}{q+1}$, d'où $u \geqslant 0$.
Par ailleurs :

$$v \geqslant r - n + (q+1)\left(1 - \frac{q+r-n}{q+1}\right) = r - n + q + 1 - q - r + n = 1.$$

Dès lors, $u = q + 1 - v \leqslant q + 1 - 1 = q$ et $v = q + 1 - u \leqslant q + 1$ d'où $u \in [\![0; q]\!]$ et $v \in [\![1; q+1]\!]$. □

Remarque. Le quotient q est supposé non nul. En effet, si $q = 0$, $N = r$ et cela peut poser problème si $n \leqslant N$: c'est exactement les entiers n que l'on considère.

La nouvelle partition associée à cette décomposition est alors :

$$\tau = (\underbrace{\alpha + 1, \ldots, \alpha + 1}_{v \text{ fois}}, \underbrace{\alpha, \ldots, \alpha}_{u \text{ fois}}).$$

Remarque. Cette décomposition est légitime et représente bien une partition. En effet, α et $\alpha + 1$, qui sont les tailles des groupes, sont majorés par N. De plus, $u + v$, qui est le nombre de groupes, vaut $q + 1$ et est majoré en pratique par $N + 1$ (c'est le quotient augmenté de 1). Il peut exister des cas où $q + 1 = N + 1$, mais alors $n = 1$ et $r = 0$, ce qui donne $\alpha = 0$ et $u = 1$: on a en fait ici N groupes.

Remarque. Les expressions de α, u et v proviennent de l'algorithme suivant, qui permet de calculer la décomposition :

```
1: fonction CombCons(n, q, r) :
2:     n' ← n
3:     r' ← r
4:     Tant que n' − r' > q, faire :
5:         n' ← n' − 1
6:         r' ← r' + q
7:     Fin tant que
8:     α ← n' − 1
9:     u ← n' − r'
```

10: $v \leftarrow q - r' + n' + 1$
11: **renvoyer** (α, u, v)
12: **Fin fonction**

Les entiers n' et q' sont en fait des suites arithmétiques. On calcule le plus petit rang tel que $n' - r' \leqslant q$ puis on en déduit les expressions ci-dessus.

Maintenant qu'on a une décomposition plus fidèle, on va la comparer avec l'ancienne.

Pour distinguer les deux compositions, on notera $\mathbb{E}(T(n)) \overset{\text{DE}}{=} \cdots$ ou $\mathbb{E}(T(n)) \overset{\text{CC}}{=} \cdots$ pour désigner respectivement l'espérance issue de la décomposition par division euclidienne, et celle issue de la décomposition en combinaison linéaire de deux entiers naturels consécutifs.

Propriété 4.3

Soit $n > 1$. Notons $N = nq + r$ la division euclidienne de N par n. Notons $N = \alpha u + (\alpha + 1)v$ la décomposition en combinaison linéaire de deux entiers consécutifs. On a :

— si $\alpha = 0$, alors nécessairement $v = N$ et $\mathbb{E}(T(n)) \overset{\text{CC}}{=} N$;

— si $\alpha = 1$, alors :

$$\boxed{\mathbb{E}(T(n)) \overset{\text{CC}}{=} u + 3v - 2v(1-p)^2} ;$$

— si $\alpha > 1$, alors :

$$\boxed{\mathbb{E}(T(n)) \overset{\text{CC}}{=} N + u + v - \big(N - (\alpha+1)vp\big)(1-p)^{\alpha}} .$$

Démonstration. Encore une fois, on applique la formule 3.1 avec $Se = Sp = 1$ dans les différents cas... □

À l'aide de ces formules, on en déduit un programme de comparaison.

Code source 4.5 : comparaison des deux décompositions, en Python

```
import matplotlib.pyplot as plt
import numpy as np


p0 = 1 - np.exp(-np.exp(-1))


def esperanceT_DE(N, n, p):
    if n == 1:
        return N
    q, r = N // n, N % n
```

```
    if r == 0:
        return N * (1 + 1/n - (1-p)**n)
    if r == 1:
        return N + q - n*q*(1-p)**n
    return N + q + 1 - n*q*(1-p)**n - r*(1-p)**r


def combCons(n, q, r):
    N = n
    R = r
    while N - R > q:
        N -= 1
        R += q
    u = N - R
    v = q + R - N + 1
    return (N - 1, u, v)


def esperanceT_CombCons(N, n, p):
    if n == 1:
        return N
    q, r = N // n, N % n
    (a, u, v) = combCons(n, q, r)
    if a == 0:
        return N
    if a == 1:
        return u + 3 * v - 2 * v * (1-p)**2
    return N + u + v - (N-(a+1)*v*p) * (1-p)**a


def esperanceT_VS_n(N, p):
    ns = np.arange(1, N+1)
    es_DE = [esperanceT_DE(N, n, p) for n in ns]
    es_CC = [esperanceT_CombCons(N, n, p) for n in ns]
    plt.clf()
    plt.axhline(N, color = "red")
    plt.plot(ns, es_DE, "+", label = "DE")
    plt.plot(ns, es_CC, "+", label = "CC")
    plt.legend()
    plt.show()
```

```
def min_fonction(f, N, p):
    min_n, min_e = 1, f(1, N, p)
    for n in range(2, N+1):
        e = f(n, N, p)
        if e < min_e:
            min_n, min_e = n, e
    return min_n


def nbEcart_NFixe(N):
    ps = np.linspace(0.001, p0, 1000)
    ecarts = {}
    for p in ps:
        min_n_espT_DE = min_fonction(esperanceT_DE, N, p)
        min_n_espT_CC = min_fonction(esperanceT_CombCons, N, p)
        e = abs(min_n_espT_DE - min_n_espT_CC)
        try:
            ecarts[e] += 1
        except KeyError:
            ecarts[e] = 1
    return sorted(ecarts.items())

esperanceT_VS_n(80, 0.2)

for N in [10, 20, 53, 80, 100, 200, 500, 1000]:
    print(N, nbEcart_NFixe(N))
```

Les courbes de la figure 9 sont globalement les mêmes, surtout pour n faible.

Avec le tracé d'autres graphes (non fait ici), on peut montrer que les seuils proches de p_0 pour la division euclidienne sont aussi des seuils valables pour cette décomposition.

On conjecture alors que le minimum des deux décompositions est le même. Pour vérifier cela, on calcule le tableau des effectifs des écarts comme précédemment. On obtient :

```
10 [(0, 1000)]
20 [(0, 1000)]
53 [(0, 1000)]
80 [(0, 1000)]
100 [(0, 1000)]
200 [(0, 1000)]
500 [(0, 1000)]
1000 [(0, 1000)]
```

On voit que le mimimum de l'espérance pour la décomposition en combinaison linéaire

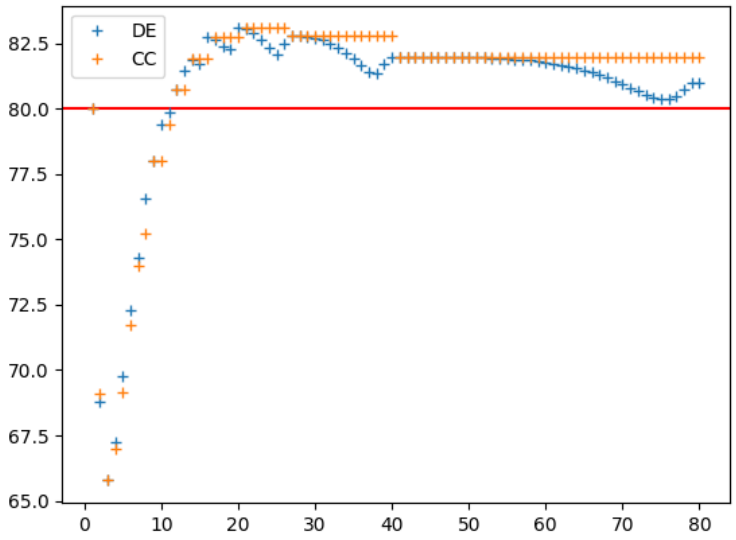


FIGURE 9 – Espérance de $T(n)$ en fonction de n pour les deux décompositions, pour $N = 80$ et $p = 0{,}2$

de deux entiers consécutifs est le même que pour la division euclidienne. En ce sens, le théorème 4.3 reste valide pour cette décomposition. On n'a pas besoin de réaliser d'autres études.

Ainsi, on choisira toujours cette décomposition, plus fidèle et donnant le même minimum et la même espérance (un inconvénient de cette décomposition est sa dépendance avec la division euclidienne : on doit d'abord passer par celle-ci avant de calculer la décomposition, rendant les programmes de calcul plus longs).

Un problème majeur de cette décomposition concerne le cas où n divise N. En effet, lorsqu'on calcule la décomposition avec $r = 0$, on n'obtient pas nécessairement $u = 0$: on transforme un cas idéal en un cas beaucoup moins idéal. En fait, comme on a vu que les minima sont les mêmes sur $[\![2; N]\!]$, cela n'a pas d'importance : lorsque n minimise l'espérance avec $r = 0$, on ne calcule pas la décomposition ; on garde les q groupes de n.

4.1.4. Résumé

Théorème 4.5 : détermination de la conception optimisant $\mathbb{E}(T(n))$ avec des groupes de même taille (à 1 près au pire) pour une population homogène, avec un test parfait

Pour une population homogène assez grande ($N \geqslant 6$), dans la procédure du test groupé avec des groupes de même taille (à 1 près) :

- si $p \geqslant \widetilde{p}_0(N) \approx p_0$, alors la procédure n'est pas optimale : il vaut mieux réaliser « classiquement » des tests individuels ;
- si $s_0(N) < p < \widetilde{p}_0(N) \approx p_0$ où $s_0(N) = 1 - \mathrm{e}^{\frac{1}{N}W_0(-\frac{1}{N})}$, on calcule $\frac{Z_0\left(-\ln(1-p)\right)}{\ln(1-p)}$, on prend l'entier n le plus proche de ce nombre, puis on calcule la division euclidienne de N par n. Ensuite, en notant q et r respectivement le quotient et le reste :
  - si $r = 0$, on prend $\tau = (\underbrace{n, \dots, n}_{q \text{ fois}})$;
  - sinon, on calcule $\textsc{CombCons}(n, q, r) = (\alpha, u, v)$, puis on en déduit
$$\tau = (\underbrace{\alpha+1, \dots, \alpha+1}_{v \text{ fois}}, \underbrace{\alpha, \dots, \alpha}_{u \text{ fois}});$$
- si $p \leqslant s_0(N)$, on prend $\tau = (N)$.

4.2. Groupes de taille quelconque

On va maintenant passer au cas général : on cherche à établir des conceptions dont la taille des groupes est quelconque, *i.e.* on regarde toutes les partitons de N.

La formule 3.1 de l'espérance de $T(\tau)$ est trop compliquée à étudier théoriquement. On va se contenter de résultats quantitatifs, en programmant un algorithme de recherche de la partition optimale.

4.2.1. Calcul des partitions d'un entier

Pour trouver la conception idéale, on va parcourir toutes les partitions possibles de N et sélectionner celle qui minimise $\mathbb{E}(T(\tau))$. Pour cela, il nous faut savoir construire ces partitions.

La méthode qui suit provient d'un devoir surveillé d'option informatique (*cf.* http://www.info-llg.fr/option-mpsi/prive/pdf/enonceb.pdf).

Définition 4.1 : ordre lexicographique inversé

On introduit la relation $\succ$ sur l'ensemble des partitions de N, appelée ordre lexicographique inversé, définie comme suit : pour deux partitions $\tau_1 = (a_1, a_2, \dots, a_k)$ et $\tau_1 = (b_1, b_2, \dots, b_l)$ (dont les éléments sont toujours ordonnés dans l'ordre

décroissant et dont leur somme vaut N), $\tau_1 \succ \tau_2$ si et seulement si il existe $m \in [\![0; \min(k,l)]\!]$ tel que $\forall i \in [\![0; m-1]\!]$, $a_i = b_i$ et $a_m > b_m$.

La relation $\succ$ est une relation d'ordre stricte, c'est-à-dire qu'elle est antiréflexive et transitive (on ne le démontre pas, c'est facile).

Exemple 4.3

our $N = 10$, on a $(4,4,2) \succ (4,2,1,1,1,1) \succ (3,3,3,1) \succ (2,2,2,2,2)$.

Grâce à cet relation, pour une partition donnée, on peut en déduire celle qui suit selon l'ordre $\succ$.

Propriété 4.4

Pour une partition donnée $\tau = (a_1, a_2, \dots, a_k)$ différente de $(\underbrace{1,1,\dots,1}_{N \text{ fois}})$, la partition de N qui suit pour l'ordre $\succ$ (c'est-à-dire la plus grande partition inférieure à τ) est, en notant $k_0 = \max\{i \in [\![1,k]\!] \,|\, a_i > 1\}$, q et r respectivement le quotient et le reste de la division euclidienne de $a_{k_0} + k - k_0$ par $a_{k_0} - 1$:

— si $r = 0$, $\tau' = (a_1, a_2, \dots, a_{k_0-1}, \underbrace{a_{k_0}-1, \dots, a_{k_0}-1}_{q \text{ fois}})$;

— si $r > 0$, $\tau' = (a_1, a_2, \dots, a_{k_0-1}, \underbrace{a_{k_0}-1, \dots, a_{k_0}-1}_{q \text{ fois}}, r)$.

Démonstration.

Analyse :

L'entier k_0 est bien défini car il existe au moins un entier de τ non égal à 1.

L'idée pour trouver la partition suivante est de diminuer de 1 le dernier élément non égal à 1, et de réorganiser les éléments après celui-ci (il est clair que si on diminue de plus, sous réserve d'existence, on pourra trouver une partition plus grande inférieure à τ).

Le problème est de choisir de quelle manière on réorganise les $a_{k_0+1}, \dots, a_k$ pour avoir la plus grande partition inférieure à τ. Intituivement, après $a_{k_0}-1$, on aimerait bien placer les éléments les plus grands possibles pour répondre au problème. Ici, il s'agit des $a_{k_0} - 1$.

Le nombre $k - k_0$ est le nombre de 1 qui terminent la partition. On prend $a_{k_0} + k - k_0$ qui est la somme des derniers éléments à partir du dernier strictement plus grand que 1 (celui qu'on diminue de 1), et à partir de cette somme, on veut la réorganiser pour avoir le plus de $a_{k_0} - 1$. C'est pourquoi on introduit sa division euclidienne par $a_{k_0} - 1$.

On place ainsi q fois des $a_{k_0} - 1$, et éventuellement le reste r qui est strictement

plus petit que $a_{k_0} - 1$ à la fin.

Synthèse :
Il nous reste à vérifier que cette partition convient bien. On va supposer par l'absure qu'il existe υ tel que $\tau \succ \upsilon \succ \tau'$. Notons $\upsilon = (b_1, \dots, b_l)$.

Comme $\tau \succ \tau'$ et que les $k_0 - 1$ premiers éléments sont les mêmes, on a $l \geqslant k_0$ et $\forall i \in [\![0; k_0 - 1]\!]$, $b_i = a_i$.
Supposons $b_{k_0} = a_{k_0}$. Alors d'après $\tau \succ \upsilon$, $l > k_0$ et $\exists\, l_0 > k_0$, $b_{l_0} < 1$; c'est absurde. Donc $b_{k_0} < a_{k_0}$.
Supposons $b_{k_0} > a_{k_0} - 1$. Alors $b_{k_0} \in]a_{k_0} - 1, a_{k_0}[$. C'est absurde. Comme $\forall i \in [\![0; k_0 - 1]\!]$, $b_i = a_i$ et comme $\upsilon \succ \tau'$, on a donc $b_{k_0} = a_{k_0} - 1$ (on ne peut avoir $b_{k_0} < a_{k_0} - 1$).
On en déduit que $l > k_0$. Et d'après $\upsilon \succ \tau'$, comme pour tout i dans $[\![k_0; k_0 + q - 1]\!]$, le i^{e} élément de τ' est $a_{k_0} - 1$, il est de même pour υ, et $l > k_0 + q - 1$. En résumé, υ est la même partition que τ' jusqu'au dernier $a_{k_0} - 1$ de τ'.
Comme $\upsilon \succ \tau'$ et comme les partitions ne peuvent être les mêmes, si $r > 0$, alors $b_{k_0+q} > r$. Mais alors la somme des élements de υ est strictement plus grande que N. Et si $r = 0$, υ contient un élément de plus que τ', rendant aussi la somme strictement plus grande que N. C'est absurde dans les deux cas.

Donc τ' bien la plus grande partition plus petite que τ. □

La plus grande partition possible est (N) (on ne peut trouver d'élément strictement plus grand que N). En partant de cette partition et en calculant les partitions suivantes jusqu'à $(\underbrace{1, 1, \dots, 1}_{N \text{ fois}})$ (qui n'admet aucune partition suivante ; c'est la plus petite), on aura trouvé toutes les partitions. On en déduit un algorithme de construction de l'ensemble des conceptions.

Écrivons cet algorithme en OCaml. On va représenter une partition par une liste dont les éléments sont les éléments inverses de la partition. Par exemple, $\tau = (8; 6; 5; 1; 1)$ est représenté par `[1;1;5;6;8]`. On choisit cette convention car il sera plus simple de construire les partitions suivantes.

On a vu que dans le calcul de la prochaine partition, on a besoin de savoir où est le dernier élément strictement supérieur à 1. On écrit d'abord la fonction `scinde` de signature `int list -> int * int list` qui à une partition `l` renvoie un couple `(nb, t)` où `nb` est le nombre de 1 qui finit dans la partition, et `t` la liste de `l` où on a enlevé tous les `1`.

On écrit ensuite la fonction `ajoute : int -> 'a -> 'a list -> 'a list` qui à un entier `n`, à un élément `x`, et à une liste `l` renvoie la liste `l` à laquelle on a ajouté `n` fois l'élément `x` au début.

Grâce à ces deux fonctions, on peut écrire la fonction `partition_suivante` de signature `int list -> int list` qui à une partition `l` renvoie la partition suivante. Dans cette fonction, on récupère d'abord le couple `(n, u) = scinde l`. On calcule ensuite le quotient et le reste `q` et `q` définis plus haut en prenant l'élément en tête `t` de `u` (il s'agit de a_{k_0}), puis on crée la partition suivante en ajoutant `q` fois `t` à la queue de `u` à l'aide

de `ajoute`. Si `r` est non nul, on le rajoute aussi.

Remarque. Cette fonction renvoie la liste vide si la partition est la plus petite. Ce sera un critère d'arrêt pour la fonction suivante.

Enfin, on en déduit la fonction `partition : int -> int list list` qui à `n` associe la liste des partitions de `n`, en calculant de proche en proche les partitions suivantes jusqu'à arriver à la liste vide.

Le code source se situe dans le paragraphe suivant.

4.2.2. Recherche de la partition optimale

Écrire la fonction `conceptionMin : int -> float -> int list * float`, qui détermine la partition optimale et retourne l'espérance de $T(\tau)$ associée en fonction de N et p n'est pas compliquée : on parcourt l'ensemble des partitions de N pour trouver celle minimisant l'espérance.

Il faut juste écrire une fonction `espT` de signature `int liste -> float -> float` qui à une partition `part` et à la probabilité `p` calcule grâce à la formule 3.1 avec $Se = Sp = 1$ l'espérance du nombre de tests.

Code source 4.6 : détermination de la conception minimale pour des groupes quelconques, en OCaml

```
(* partitions *)
let scinde l =
  let rec compter n l =
    match l with
    | 1 :: t -> compter (n+1) t
    | l -> (n, l)
  in
  compter 0 l ;;


let rec ajoute n x l =
  match n with
  | 0 -> l
  | n -> x :: (ajoute (n-1) x l) ;;


let partition_suivante l =
  let (n, u) = scinde l in
  match u with
  | [] -> []
  | t :: v -> begin
      let q = (n + t) / (t - 1)
```

```
      and r = (n + t) mod (t - 1) in
      match r with
      | 0 -> ajoute q (t-1) v
      | _ -> r :: (ajoute q (t-1) v)
    end ;;


let partitions n =
  let rec cons l =
    let c = partition_suivante (List.hd l) in
    match c with
    | [] -> l
    | _ -> cons (c::l)
  in
  cons [[n]] ;;


(* conception optimale *)
let espT part p =
  let q = List.length part in
  let rec somme l = match l with
    | [] -> Float.of_int q
    | a :: t when a > 1 -> (
        let n = Float.of_int a in
        let s = n *. (1. -. (1. -. p) ** n) in
        s +. (somme t)
      )
    | _ :: t -> somme t
  in
  somme part ;;


let conceptionMin nPop p =
  let listePartitions = partitions nPop in
  let rec trouverMin l minAct pAct = match l with
    | [] -> (pAct, minAct)
    | part :: t -> (
        let e = espT part p in
        if e < minAct then trouverMin t e part
        else trouverMin t minAct pAct
      )
```

```
in
let pr = List.hd listePartitions in
let nPop_f = Float.of_int nPop in
trouverMin listePartitions nPop_f pr ;;
```

Le problème de cette recherche algorithmique est sa complexité.

Propriété 4.5 : équivalent du nombre de partitions

Soit $p(N)$ le nombre de partitions de l'entier N. Alors :

$$p(N) \underset{N\to+\infty}{\sim} \frac{1}{4N\sqrt{3}} \exp\left(\pi\sqrt{\frac{2N}{3}}\right).$$

Cette formule a été démontré par Hardy et Ramanujan en 1918. En conséquence, notre fonction de calcul de la partition optimale a une complexité temporelle en :

$$O\left(\frac{1}{N}\exp\left(\pi\sqrt{\frac{2N}{3}}\right)\right).$$

Cela est moins pire qu'une complexité exponentielle, mais c'est pire qu'une complexité polynomial (par croissances comparées). Les calculs sont plutôt longs. Voici le temps d'exécution de `partitions` pour quelques valeurs de N.

N	$p(N)$	temps d'exécution
10	42	0,12 ms
20	627	0,54 ms
30	5 604	5,8 ms
40	37 338	0,14 s
50	204 226	0,98 s
60	966 467	5,1 s
70	4 087 968	23 s
80	15 796 476	1 min 08 s
90	56 634 173	6 min 05 s
100	190 569 292	13 min 41 s

Remarque. Pour calculer le temps d'exécution pour un `n` donné, on a utilisé le programme suivant :

```
let t = Sys.time() in
let _ = partitions n in
Sys.time () -. t ;;
```

Pour $N > 80$, les temps d'exécution ne sont pas raisonnables ; et encore, cela concerne seulement `partitions` alors que le calcul de la conception optimale nécessite après avoir calculé la liste des partitions de parcourir cette liste ! Par exemple, pour `conceptionMin 50 0.2`, on obtient un temps d'exécution de 1,1 s contre 0,98 s pour `partitions 50`.

Comme on pouvait s'y attendre, cette solution algorithmique n'est pas efficace. Dans la suite, on va analyser qualitativement les partitions optimales pour éventuellement optimiser notre algorithme.

4.2.3. Propriétés des partitions optimales

Regardons quelques résultats donnés par la fonction précédente.

```
# conceptionMin 40 0.1 ;;
- : int list * float = ([4; 4; 4; 4; 4; 4; 4; 4; 4; 4], 23.7559999999999931)
# conceptionMin 40 0.2 ;;
- : int list * float =
([3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 4], 32.9295999999999864)
# conceptionMin 40 0.3 ;;
- : int list * float =
([1; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3], 39.6230000000000047)
# conceptionMin 40 0.4 ;;
- : int list * float =
([1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1],
 40.)
# conceptionMin 40 0.5 ;;
- : int list * float =
([1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1],
 40.)

# conceptionMin 45 0.01 ;;
- : int list * float = ([11; 11; 11; 12], 8.81721914886880853)
# conceptionMin 45 0.1 ;;
([4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 5], 26.8035499999999907)
# conceptionMin 45 0.2 ;;
- : int list * float =
([3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3], 36.9599999999999795)

# conceptionMin 50 0.1 ;;
- : int list * float =
([3; 3; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4], 29.7575999999999858)
# conceptionMin 50 0.2 ;;
- : int list * float =
```

```
([2; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3], 41.143999999999977)
# conceptionMin 50 0.3 ;;
- : int list * float =
([1; 1; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3], 49.5360000000000298)
# conceptionMin 50 0.4 ;;
- : int list * float =
([1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1],
 50.)
```

La première chose qu'on peut remarquer est que la plupart des conceptions optimales ont l'air être des combinaisons consécutives. La deuxième chose est qu'il semble exister aussi des seuils au-dessus desquels la procédure n'est pas optimale (lorqu'on obtient que des 1, *i.e.* lorsque la meilleure conception correspond au cas classique).

Intéressons nous d'abord aux seuils d'optimalité. On devine de proche en proche les seuils :

```
# conceptionMin 40 0.306639 ;;
- : int list * float =
([1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1],
 40.)
# conceptionMin 40 0.306638 ;;
- : int list * float =
([1; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3], 39.9999591838253536)
# conceptionMin 50 0.306639 ;;
- : int list * float =
([1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1],
 50.)
# conceptionMin 50 0.306638 ;;
- : int list * float =
([1; 1; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3],
 49.9999497647081341)
```

Comme précédemment, ces seuils ont l'air proches de p_0, mais ne sont pas exactement p_0. On a cependant quelque chose de nouveau : on remarque curieusement que les seuils semblent ne pas dépendre de N, et valent tous environ 0,306 638 5. On va confirmer ce phénomène informatiquement ; c'est un programme qu'on a déjà fait, mais en Python. Voici le code en OCaml.

Code source 4.7 : détermination du seuil d'optimalité, en OCaml

```
let seuilOpt nPop nbDecimales =
  let rec seuil p nbDecimales =
    let partMin = fst (conceptionMin nPop p) in
```

```
    let l = List.length partMin in
    if l < nPop then (
      let new_p = p +. 10. ** (-. (Float.of_int nbDecimales)) in
      seuil new_p nbDecimales
    )
    else p -. 10. ** (-. (Float.of_int nbDecimales))
  in
  let rec calcul nb p0 =
    match nb with
    | nb when nb > nbDecimales -> p0
    | nb -> let new_p = seuil p0 nb in
      calcul (nb + 1) new_p
  in
  calcul 1 0. ;;
```

On a calculé `seuilOpt n 10` pour `n` entre 3 et 50. On a toujours obtenu la valeur 0,306 638 725 6. On notera p_1 ce seuil. On obtient aussi le même résultat pour 60 et on n'a pas cherché à aller au delà compte rendu des temps d'exécution de `partitions`.

Remarque. Le cas $n = 2$ est particulier puisque `seuilOpt 2 8` retourne 0,292 893 21.

Propriété 4.6 : seuil d'optimalité pour des groupes quelconques

Si $N \geqslant 3$ et si $p > p_1 \approx 0{,}306\,638\,725\,6$, alors la procédure n'est pas optimale.

Remarque. En toute rigueur, on ne sait pas ce qui se passe au delà de $N > 80$.

On posera dans la suite dans les programmes en OCaml la variable globale p_1 avec l'instruction `let p1 = 0.3066387256 ;;`.

Ce résultat étant établi, on va maintenant étudier l'autre phénomène observé précédemment : la plupart des conceptions idéales semblent être des combinaisons consécutives, au sens large (plus précisément des conceptions composées de groupes dont les tailles ne diffèrent au pire que de 1). Regardons plus précisément des conceptions idéales pour N fixé :

```
# conceptionMin 35 0.1 ;;
- : int list * float = ([3; 4; 4; 4; 4; 4; 4; 4; 4], 20.8178)
# conceptionMin 35 0.2 ;;
- : int list * float =
([2; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3], 28.8239999999999839)
# conceptionMin 35 0.25 ;;
- : int list * float = ([2; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3], 31.953125)
# conceptionMin 35 0.29 ;;
- : int list * float =
([2; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3], 34.1807369999999935)
```

```
# conceptionMin 35 0.3 ;;
- : int list * float = ([1; 1; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3], 34.681)

# conceptionMin 25 0.1 ;;
- : int list * float = ([4; 4; 4; 4; 4; 5], 14.9255500000000012)
# conceptionMin 25 0.2 ;;
- : int list * float = ([3; 3; 3; 3; 3; 3; 3; 4], 20.6095999999999897)
# conceptionMin 25 0.24 ;;
- : int list * float = ([3; 3; 3; 3; 3; 3; 3; 4], 22.4470169599999956)
# conceptionMin 25 0.25 ;;
- : int list * float = ([1; 3; 3; 3; 3; 3; 3; 3; 3], 22.875)
# conceptionMin 25 0.28 ;;
- : int list * float = ([1; 3; 3; 3; 3; 3; 3; 3; 3], 24.0420479999999941)
# conceptionMin 25 0.3 ;;
- : int list * float = ([1; 3; 3; 3; 3; 3; 3; 3; 3], 24.768)
```

Avec d'autres d'exemples, on conjecture le résultat suivant : il existe un seuil, inférieur à p_1, tel que pour p en dessous, les partitions optimales sont des combinaisons de deux entiers ; et pour p entre ce seuil et p_1, les conceptions optimales ne sont plus des combinaisons consécutives. On va montrer ce résultat informatiquement.

Écrivons d'abord une fonction `estCC` vérifiant si une liste représentant une partition est composée de seulement deux entiers consécutifs. On en déduit ensuite la fonction `seuilCC`, du même modèle que `seuilOpt`, qui calcule le seuil désiré.

Code source 4.8 : calcul des seuils de probabilités au delà desquels les conceptions optimales ne sont plus des combinaisons consécutives, en OCaml

```
let estCC l =
  let rec listeCste l =
    match l with
    | [] -> true
    | [a] -> true
    | a :: b :: t  when a = b -> listeCste (b :: t)
    | _ -> false
  in
  let rec chercherDeuxieme l =
    match l with
    | [] -> true
    | [a] -> true
    | a :: b :: t when a = b -> chercherDeuxieme (b :: t)
    | a :: b :: t when b = a + 1 -> listeCste (b :: t)
    | _ -> false
  in
```

```
  chercherDeuxieme l ;;


let calculer_seuilCC nPop nbDecimales =
  let rec seuil p nbDecimales =
    let partMin = fst (conceptionMin nPop p) in
    if p <= p1 && estCC partMin then (
      let new_p = p +. 10. ** (-. (Float.of_int nbDecimales)) in
      seuil new_p nbDecimales
    )
    else p -. 10. ** (-. (Float.of_int nbDecimales))
  in
  let rec calcul nb p0 =
    match nb with
    | nb when nb > nbDecimales -> p0
    | nb -> let new_p = seuil p0 nb in
      calcul (nb + 1) new_p
  in
  calcul 1 0. ;;
```

Voici quelques résultats :

```
# calculer_seuilCC 20 10 ;;
- : float = 0.292893218800000299
# calculer_seuilCC 30 10 ;;
- : float = 0.3066387256
# calculer_seuilCC 40 10 ;;
- : float = 0.249999999900000047
```

On obtient 3 seuils différents. En fait, ce sont les seuls. En effet, voici les seuils obtenus dans l'ordre pour N allant de 2 à 30, et pour 5 décimales :

```
0.30663
0.30663
0.24999
0.29289
0.30663
0.24999
0.29289
0.30663
0.24999
0.29289
0.30663
0.24999
```

```
0.29289
0.30663
0.24999
0.29289
0.30663
0.24999
0.29289
0.30663
0.24999
0.29289
0.30663
0.24999
0.29289
0.30663
0.24999
0.29289
0.30663
```

À part $N = 2$, on remarque que ces seuils forment une suite périodique de période 3 (on fait cela pour des décimales plus grandes ; on a en pris 5 pour rendre plus clair le résultat). De plus, l'un de ces seuils est exactement p_1. On en déduit le théorème suivant :

Propriété 4.7

Pour $N \geqslant 3$, on a :
- si $N \equiv 0\,[3]$, toutes les partitions optimales pour $p \leqslant p_1$ sont des combinaisons consécutives ;
- si $N \equiv 1\,[3]$, les partitions optimales pour p entre $p_{\equiv 1} \approx 0{,}249\,999\,99$ (exclu) et p_1 (inclu) ne sont pas des combinaisons consécutives, et toutes celles pour $p \leqslant p_1$ sont le sont ;
- si $N \equiv 2\,[3]$, les partitions optimales pour p entre $p_{\equiv 2} \approx 0{,}292\,893\,21$ (exclu) et p_1 (inclu) ne sont pas des combinaisons consécutives, et toutes celles pour $p \leqslant p_1$ sont le sont.

On a même encore mieux. En analysant les résultats pour les probabilités donnant des combinaisons non consécutives, on remarque :

Propriété 4.8

Pour $N \geqslant 3$, on a :
- si $N \equiv 1\,[3]$ et $p_{\equiv 1} < p \leqslant p_1$ alors la partition optimale est, en notant q le

reste de la division euclidienne de N par 3 :

$$\tau = (\underbrace{3, 3, \ldots, 3}_{q \text{ fois}}, 1);$$

— si $N \equiv 2\,[3]$ et $p_{\equiv 2} < p \leqslant p_1$ alors la partition optimale est :

$$\tau = (\underbrace{3, 3, \ldots, 3}_{q \text{ fois}}, 1, 1).$$

On va maintenant se concentrer et se placer sur les intervalles où les conceptions optimales sont des combinaisons consécutives.

On peut être amené à se poser la question suivante : si dans ces cas, les meilleurs conceptions sont des combinaisons consécutives, celles-ci seraient-elles les mêmes que dans l'étude précédente où les groupes sont de même taille (à 1 près au pire) ?

On va regarder cela avec des fréquences (comme pour le cas mêmes groupes). Il nous faut d'abord écrire la fonction `conceptionMemeGroupe` qui donne la meilleure partition selon le théorème 4.5. On a donc besoin des fonctions W_0 et Z_0 : on les programme à l'aide de la méthode de dichotomie (ce sont des solutions implicites d'équations). Il nous faut aussi la fonction CombCons.

Code source 4.9 : calcul de la fréquence des conceptions idéales identiques entre le cas quelconque et le cas même taille, en OCaml

```
(* fonctionZ *)
let dicho f a b eps =
  let rec reduire_recherche a b =
    let c = (a +. b) /. 2. in
    if b -. a > eps then (
      if (f a) *. (f c) <= 0. then
        reduire_recherche a c
      else reduire_recherche c b
    )
    else c
  in
  reduire_recherche a b ;;


let reciZ0 y =
  assert ( 0. < y && y < 4. *. (exp (-2.)) ) ;
  let z_y = fun x -> x ** 2. *. (exp x) -. y in
  dicho z_y (-2.) 0. (1e-8) ;;
```

```
(* fonction W0 de Lambert *)
let reciW0_neg y =
  assert ( -. (exp (-1.)) < y && y < 0. ) ;
  let w_y = fun x -> x *. (exp x) -. y in
  dicho w_y (-1.) 0. (1e-8) ;;


(* seuil s0(N) *)
let seuil0 nPop =
  let nPop_f = Float.of_int nPop in
  let w = reciW0_neg (-. 1. /. nPop_f) in
  1. -. (exp (w /. nPop_f)) ;;


(* CombCons *)
let comb_cons n q r =
  let rec calcul n r =
    if n - r > q then
      calcul (n - 1) (r + q)
    else (n, r)
  in
  let (np, rp) = calcul n r in
  let alpha = np - 1
  and u = np - rp
  and v = q + rp - np + 1 in
  (alpha, u, v) ;;


(* Construction liste de mêmes éléments *)
let rec listeMemesEl nb el =
  match nb with
  | 0 -> []
  | nb -> el :: (listeMemesEl (nb - 1) el) ;;


(* Calcul de la conception dans le cas "Même taille" *)
let conceptionMemeTaille nPop p =
  match p with
  | p when p <= seuil0 nPop -> [nPop]
```

```
  | p when p >= 1. -. exp(-.exp (-1.)) -> listeMemesEl nPop 1
  | p -> begin
      let n =
        let argu = -. log (1. -. p) in
        Float.round ((reciZO argu) /. (-. argu)) |> Float.to_int
      in
      let q = nPop / n and r = nPop mod n in
      if r = 0 then listeMemesEl q n
      else (
        let (alpha, u, v) = comb_cons n q r in
        let t1 = listeMemesEl u alpha
        and t2 = listeMemesEl v (alpha+1) in
        t1 @ t2
      )
    end ;;


(* Comparaison cas même taille / taille quelconque *)
let comparer nPop p =
  let c_qcq = fst (conceptionMin nPop p)
  and c_mt = conceptionMemeTaille nPop p in
  c_qcq = c_mt ;;


(* Seuils pour CC *)
let seuilCC nPop =
  match nPop with
  | nPop when nPop mod 3 = 0 -> p1
  | nPop when nPop mod 3 = 1 -> 0.2499999999
  | _ -> 0.2928932188 ;;


let frequenceMemeConc nPop =
  let rec comptage ps =
    match ps with
    | [] -> 0
    | p :: t when comparer nPop p -> 1 + comptage t
    | _ :: t -> comptage t
  in
  let f i =
    let s = seuilCC nPop in
```

```
    let i_f = Float.of_int i in s *. (i_f +. 1.) /. 100.
  in
  let nb = List.init 100 f |> comptage in
  (Float.of_int nb) /. 100. ;;
```

On obtient les résultats suivants :

```
# frequenceMemeConc 25 ;;
- : float = 0.22
# frequenceMemeConc 30 ;;
- : float = 0.88
# frequenceMemeConc 40 ;;
- : float = 0.43
# frequenceMemeConc 50 ;;
- : float = 0.7
```

On n'arrive pas à voir un schéma quelconque : on ne peut rien dire quant à la proximité des conceptions idéales dans les deux cas, si ce n'est que la partition optimale dans le cas quelconque n'est généralement pas celle calculée dans le cas même taille.

Un moyen d'optimiser notre parcours des partitions est ainsi de ne regarder que les combinaisons consécutives. Comme on vient juste de le voir, on ne peut se contenter du cas même groupe. En fait, on va montrer que dans certains cas, une conception optimale en combinaisons consécutive ne correspond à aucun antécédent de CombCons).

L'algorithme qui suit prend en paramètres N et p puis détermine si la conception optimale est un antécédent de CombCons ou non. Dans le cas positif, il renvoie l'entier n associé.

Code source 4.10 : calcul, s'il existe, de l'antécédent de CombCons pour la partition optimale, en OCaml

```
let trouverN_CC nPop p =
  let c_opt = fst (conceptionMin nPop p) in
  let rec testerCC n =
    match n with
    | n when n > nPop -> failwith "il n'existe pas d'entier qui
    ↪  convient"
    | n -> let q = nPop / n and r = nPop mod n in
      let (alpha, u, v) = comb_cons n q r in
      let t1 = listeMemesEl u alpha
      and t2 = listeMemesEl v (alpha+1) in
      if t1 @ t2 = c_opt then n
      else testerCC (n+1)
  in
```

```
testerCC 1 ;;
```

Voici quelques résultats illustrant ce qu'on veut montrer.

```
# trouverN_CC 20 0.1 ;;
- : int = 5
# trouverN_CC 20 0.2 ;;
- : int = 3
# trouverN_CC 30 0.1 ;;
- : int = 4
# trouverN_CC 30 0.2 ;;
Exception: Failure "il n'existe pas d'entier qui convient".
# trouverN_CC 40 0.1 ;;
Exception: Failure "il n'existe pas d'entier qui convient".
# trouverN_CC 40 0.2 ;;
Exception: Failure "il n'existe pas d'entier qui convient".
```

Ainsi, il nous faut un nouveau moyen de calcul de l'ensemble des combinaisons consécutives pour optimiser le parcours des partitions.

4.2.4. Calcul de l'ensemble des combinaisons consécutives

En regardant l'ensemble des partitions pour N faible, on conjecture qu'il y a exactement N combinaisons consécutives et que parmi elles, pour $s \in [\![1; N]\!]$, il en existe une unique de taille s (avec s éléments). Montrons cette conjecture.

Soit $s \in [\![1; N]\!]$. Écrivons la division euclidienne de N par s :

$$\exists!\, q \in \mathbb{N}^*,\ \exists!\, r \in [\![0; s-1]\!],\ N = qs + r.$$

On écrit d'une autre manière la somme : $N = q(s-r) + (q+1)r$. Il s'agit d'une combinaison consécutive de deux entiers de N, correspondant bien à une partition car $s - r \in [\![1; N]\!]$ et $r \in [\![0; N-1]\!]$. Et il y a s termes car $(s-r) + r = s$ (même si $r = 0$). Par unicité de q et r, cette décomposition est unique à s.

Comme s varie de 1 à N et que les combinaisons sont composées d'un nombre de termes distincts, on a trouvé N combinaisons consécutives distinctes. Et par unicité pour s fixé, on ne peut en trouver d'autres.

Remarque. Avant, c'était le terme v qui était non nul. Ici, c'est u. À vrai dire, cela n'a pas trop d'importance car la définition d'une conbinaison consécutive est au sens large (s'il n'y a qu'un seul terme dans la partition, on considère que c'est quand même une combinaison consécutive).

Remarque. Fondamentalement, il ne faut pas confondre les deux constructions des combinaisons consécutives présentées. L'algorithme CombCons prend en argument n qui est la taille des groupes que l'on souhaite avoir. À l'issue du calcul, le nombre de groupes reste le même (*cf.* la démonstration de l'algorithme ; on a $u+v = q+1$). Cependant, pour

les décompositions ci-dessus, on se fixe non plus n mais s qui est le nombre de groupes que l'on veut à l'issue du calcul. C'est pourquoi on n'a pas utilisée cette dernière méthode la première fois car on ne cherchait pas à faire varier le nombre de groupes mais juste à réorganiser les tailles des groupes.

Ce qui précède nous donne un moyen explicite de calcul des partitions : on peut réduire notre parcours des partitions à seulement N d'entre elles, ce qui est beaucoup mieux !

Voici un algorithme qui calcule la partition optimale et son espérance associée, qui prend en compte tous les cas possibles pour p :

Code source 4.11 : détermination plus efficace de la partition optimale, en OCaml

```
(* calcul de toutes les CC *)
let ensembleCC nPop =
  let rec calcul s =
    match s with
    | 0 -> []
    | s -> (
        let q = nPop / s and r = nPop mod s in
        let t1 = listeMemesEl (s-r) q
        and t2 = listeMemesEl r (q+1) in
        (t1 @ t2) :: calcul (s-1)
      )
  in
  calcul nPop ;;


(* Conception minimale pour p au dessus du seuil CC *)
let conceptionMinNonCC nPop =
  match nPop with
  | nPop when nPop mod 3 = 0 -> listeMemesEl nPop 1
  | nPop when nPop mod 3 = 1 -> (
      let q = nPop / 3 in 1 :: (listeMemesEl q 3)
    )
  | _ -> (
      let q = nPop / 3 in 1 :: 1 :: (listeMemesEl q 3)
    ) ;;


(* calcul de la conception optimale pour les CC *)
let conceptionMinEff nPop p =
  let s = seuilCC nPop in
  match p with
```

```
  | p when p > s && p <= p1 -> (
      let part = conceptionMinNonCC nPop in
      let e = espT part p in
      (part, e)
    )
  | p when p > p1 -> (List.init nPop (fun i -> 1), Float.of_int
  ↪  nPop)
  | p -> (
      let listeCC = ensembleCC nPop in
      let rec trouverMin l minAct pAct = match l with
        | [] -> (pAct, minAct)
        | part :: t -> (
            let e = espT part p in
            if e < minAct then trouverMin t e part
            else trouverMin t minAct pAct
          )
      in
      let pr = List.hd listeCC in
      let nPop_f = Float.of_int nPop in
      trouverMin listeCC nPop_f pr
    ) ;;
```

On vérifie que l'on a bien les mêmes partitions optimales :

```
# conceptionMin 20 0.2 ;;
- : int list * float = ([2; 3; 3; 3; 3; 3; 3], 16.5039999999999907)
# conceptionMinEff 20 0.2 ;;
- : int list * float = ([2; 3; 3; 3; 3; 3; 3], 16.5039999999999907)
# conceptionMin 40 0.28 ;;
- : int list * float =
([1; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3], 38.443328)
# conceptionMinEff 40 0.28 ;;
- : int list * float =
([1; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3], 38.443328)
# conceptionMin 60 0.1 ;;
- : int list * float =
([4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4], 35.633999999999979)
# conceptionMinEff 60 0.1 ;;
- : int list * float =
([4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4], 35.633999999999979)
```

On peut calculer la meilleure partition pour N beaucoup plus grand que 80 maintenant. Par exemple, `conceptionMinEff 2000 0.2` renvoie un résultat en environ 0,40 s

(cela correspond à un peu plus que le temps de calcul de l'ensemble des partitions pour $N = 40$).

Cet algorithme conclut l'étude d'une population homogène avec un test parfait. Dans la suite, on va rendre le test imparfait.

5. Test non parfait

On va reprendre la même démarche que précédemment. Cependant, le test n'est cette fois-ci pas parfait : Se et Sp ne valent pas tous les deux 1.

5.1. Groupes de même taille

Cherchons dans un premier temps à former des groupes de même taille.

Bien que la nature du test change, la manière de concevoir les groupes reste la même. On va alors reprendre les deux décompositions que ci-dessus, à savoir celle issue de la division euclidienne et celle issue de la combinaison de deux entiers consécutifs. On reprend ainsi la même notation $T(n)$ que dans le cas sans bruit, et on notera toujours pour distinguer les deux décompositions $\mathbb{E}\big(T(n)\big) \overset{\text{DE}}{=} \cdots$ ou $\mathbb{E}\big(T(n)\big) \overset{\text{CC}}{=} \cdots$.

Propriété 5.1

Soit n un entier dans $[\![2; N]\!]$. Notons $N = nq + r$ la division euclidienne de N par n. On a :

— si $r = 0$, alors :

$$\boxed{\mathbb{E}\big(T(n)\big) \overset{\text{DE}}{=} N\left(Se - \frac{1}{n} + (Se + Sp - 1)(1-p)^n\right)};$$

— si $r = 1$, alors :

$$\boxed{\mathbb{E}\big(T(n)\big) \overset{\text{DE}}{=} q + 1 + qn\big(Se - (Se + Sp - 1)(1-p)^n\big)};$$

— si $r > 1$, alors :

$$\boxed{\mathbb{E}\big(T(n)\big) \overset{\text{DE}}{=} NSe + q + 1 - (Se + Sp - 1)\big(qn(1-p)^n + r(1-p)^r\big)}.$$

Propriété 5.2

Soit $n > 1$. Soit $N = \alpha u + (\alpha + 1)v$ la décomposition en combinaison linéaire de deux entiers consécutifs. On a :

— si $\alpha = 0$, alors nécessairement $v = N$ et $\mathbb{E}\big(T(n)\big) \overset{\text{CC}}{=} N$;

— si $\alpha = 1$, alors :

$$\boxed{\mathbb{E}(T(n)) \overset{\text{CC}}{=} u + v + 2v\left(Se - (Se + Sp - 1)(1-p)^2\right)};$$

— si $\alpha > 1$, alors :

$$\boxed{\mathbb{E}(T(n)) \overset{\text{CC}}{=} NSe + u + v - (Se + Sp - 1)(N - (\alpha+1)vp)(1-p)^{\alpha}}.$$

Remarque. On retouve bien sûr les mêmes formules que précédemment si $Se = Sp = 1$.

On veut maintenant voir si les résultats du cas sans bruit restent valides avec des tests non parfaits. On veut notamment savoir si :
— il existe des seuils d'optimalité ;
— le minimum de l'espérance de $T(n)$ pour la division euclidienne est le même que dans le cas idéal ;
— le minimum de l'espérance de $T(n)$ pour la décomposition en somme de deux entiers consécutifs est le même que pour la division euclidienne.

5.1.1. Proximité entre les deux décompositions

Commençons par tracer l'espérance de $T(n)$ en fonction de n pour les deux méthodes. On se fixe une population N, une probabilité p, une sensibilité Se et une spécificité Sp. On obtient les courbes ci-contre (10, 11 et 12).

Code source 5.1 : programme de tracé des courbes 10, 11 et 12, en Python

```
import matplotlib.pyplot as plt
import numpy as np


def f(n, N, p, Se, Sp):
    return N * (Se + 1/n - (Se+Sp-1)*(1-p)**n)


def esperanceT_DE(n, N, p, Se, Sp):
    if n == 1:
        return N
    q, r = N // n, N % n
    if r == 0:
        return f(n, N, p, Se, Sp)
    if r == 1:
```

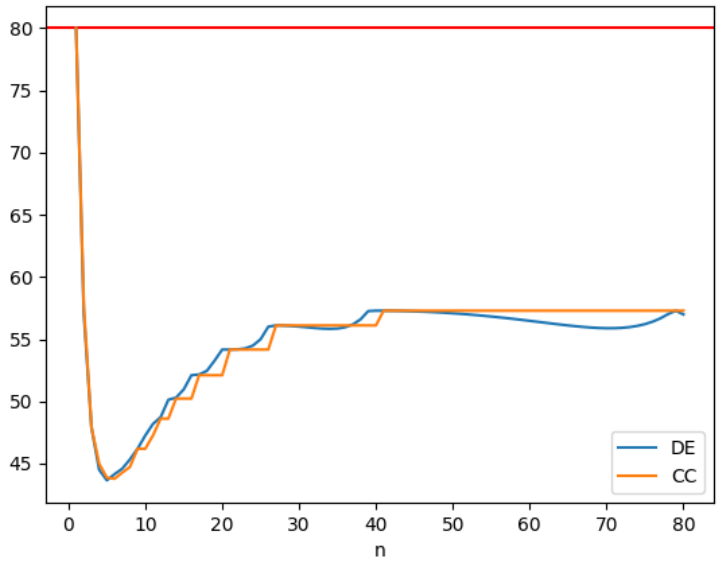


FIGURE 10 – Espérance de $T(n)$ en fonction de n pour les deux décompositions, pour $N = 80$, $p = 0{,}1$, $Se = 0{,}7$ et $Sp = 0{,}9$

```
        return q + 1 + q*n*(Se - (Se+Sp-1)*(1-p)**n)
    return N*Se + q + 1 - (Se+Sp-1)*(q*n*(1-p)**n + r*(1-p)**r)


def combCons(n, q, r):
    N = n
    R = r
    while N - R > q:
        N -= 1
        R += q
    u = N - R
    v = q + R - N + 1
    return (N - 1, u, v)


def esperanceT_CC(n, N, p, Se, Sp):
    if n == 1:
        return N
    q, r = N // n, N % n
```

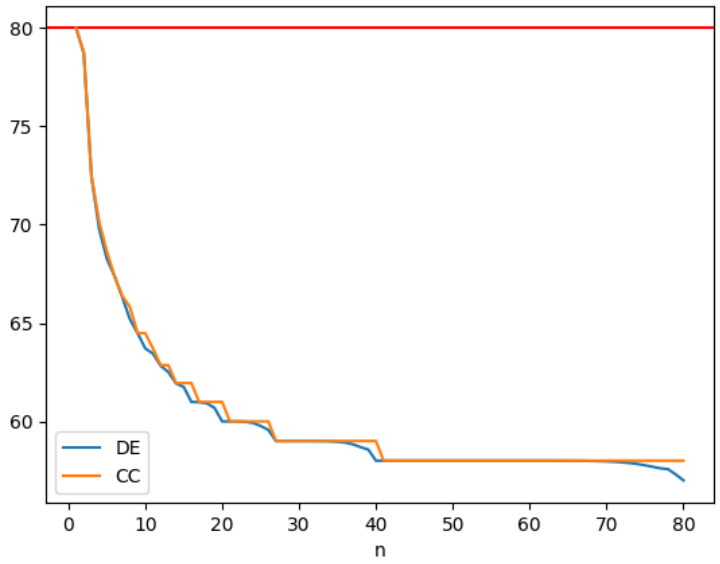


FIGURE 11 – Espérance de $T(n)$ en fonction de n pour les deux décompositions, pour $N = 80$, $p = 0{,}4$, $Se = 0{,}7$ et $Sp = 0{,}9$

```
    (a, u, v) = combCons(n, q, r)
    if a == 0:
        return N
    if a == 1:
        return u + v + 2*v*(Se - (Se+Sp-1)*(1-p)**2)
    return N*Se + u + v - (Se+Sp-1) * (N-(a+1)*v*p) * (1-p)**a


def comp_DE_CC(N, p, Se, Sp):
    ns = np.arange(1, N+1)
    de = [esperanceT_DE(n, N, p, Se, Sp) for n in ns]
    cc = [esperanceT_CC(n, N, p, Se, Sp) for n in ns]
    plt.clf()
    plt.axhline(N, color = "red")
    plt.plot(ns, de, label = "DE")
    plt.plot(ns, cc, label = "CC")
    plt.xlabel("n")
    plt.legend()
    plt.show()
```

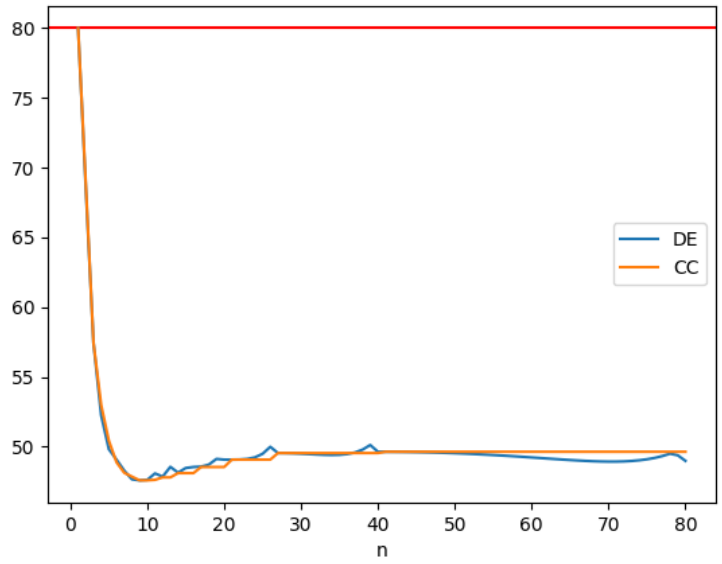


FIGURE 12 – Espérance de $T(n)$ en fonction de n pour les deux décompositions, pour $N = 80$, $p = 0{,}1$, $Se = 0{,}6$ et $Sp = 0{,}7$

```
comp_DE_CC(80, 0.1, 0.7, 0.9)
comp_DE_CC(80, 0.4, 0.7, 0.9)
comp_DE_CC(80, 0.1, 0.6, 0.7)
```

Les deux espérances sont globalement les mêmes. Pour vérifier que la proximité est bien légitime dans le cas général, réalisons une étude statistique sur l'écart entre les minima des deux décompositions pour différentes valeurs de p (on fixe N, Se, Sp)

Code source 5.2 : étude statistique des écarts entre les minimums des deux méthodes, en Python

```
import numpy as np


def f(n, N, p, Se, Sp):
    return N * (Se + 1/n - (Se+Sp-1)*(1-p)**n)
```

```
def espT_DE(n, N, p, Se, Sp):
    if n == 1:
        return N
    q, r = N // n, N % n
    if r == 0:
        return f(n, N, p, Se, Sp)
    if r == 1:
        return q + 1 + q*n*(Se - (Se+Sp-1)*(1-p)**n)
    return N*Se + q + 1 - (Se+Sp-1)*(q*n*(1-p)**n + r*(1-p)**r)


def combCons(n, q, r):
    N = n
    R = r
    while N - R > q:
        N -= 1
        R += q
    u = N - R
    v = q + R - N + 1
    return (N - 1, u, v)


def espT_CC(n, N, p, Se, Sp):
    if n == 1:
        return N
    q, r = N // n, N % n
    (a, u, v) = combCons(n, q, r)
    if a == 0:
        return N
    if a == 1:
        return u + v + 2*v*(Se - (Se+Sp-1)*(1-p)**2)
    return N*Se + u + v - (Se+Sp-1) * (N-(a+1)*v*p) * (1-p)**a


def minEspT(N, p, Se, Sp, methode):
    min_n, min_e = 1, N
    for n in range(2, N+1):
        e = methode(n, N, p, Se, Sp)
        if e <= min_e:
            min_n, min_e = n, e
    return min_n
```

```python
def statsEcart(N, Se, Sp):
    ps = np.linspace(0.01, 0.99, 100)
    ecarts = [abs(minEspT(N, p, Se, Sp, espT_DE) - minEspT(N, p,
    ↪  Se, Sp, espT_CC)) for p in ps]
    med = np.median(ecarts)
    q1, q3 = np.quantile(ecarts, 0.25), np.quantile(ecarts, 0.75)
    d9 = np.quantile(ecarts, 0.9)
    return (q1, med, q3, d9)
```

Remarque. Dans le calcul du minimum, la condition pour que `n` prenne la valeur `min_n` contient un inférieur ou égal au lieu d'un strict. En fait, la combinaison consécutive devient constante pour n assez grand, et dans certains cas, le minimum est atteint dans ces entiers. On a donc une multitude de minima possibles tandis que pour la division euclidienne, ce minimum est unique. Pour éviter d'avoir un grand écart alors qu'il n'a pas lieu d'être, une solution est de rendre l'inégalité large. Cela a pour conséquence de rendre le minimum égal à N, ce qui est cohérent avec les courbes obtenues ci-dessus (dans la zone où l'espérance pour la combinaison consécutive est constante, celle pour la division euclidienne décroît globalement).

On obtient les résultats suivants :

```
>>> statsEcart(80, 0.7, 0.9)
(0.0, 0.0, 0.0, 0.10000000000000853)
>>> statsEcart(80, 0.6, 0.7)
(0.0, 0.0, 0.0, 1.0)
>>> statsEcart(200, 0.9, 0.8)
(0.0, 0.0, 0.0, 0.0)
>>> statsEcart(200, 0.6, 0.5)
(0.0, 0.0, 0.0, 0.0)
>>> statsEcart(1000, 0.65, 0.75)
(0.0, 0.0, 0.0, 0.0)
```

On a décidé d'utiliser la médiane, les quartiles et les déciles car la moyenne et l'écart type sont très sensibles aux valeurs extrêmes (on aurait obtenu des nombres beaucoup trop élevées pour les données qu'ils sont sensés représenter ; par exemple, pour le premier résultat ci-dessus, la moyenne est 1,6 alors que 90% des valeurs valent 0 !).

On voit que le minimum est le même (écart de 0) pour un grand nombre de probabilité. La proximité est donc bien légitime, mais pas parfaite, comme le montre la valeur moyenne de 1,6 : il existe quelques valeurs de p telles que l'écart est important. Leur nombre reste cependant bien négligeable (elles apparaîssent généralement pour $N \lessapprox 100$).

Comme avant, on va donc se limiter à l'étude de la décomposition par la division

euclidienne. On dira ensuite que les résultats sont aussi valables pour la combinaison consécutive qui, rappelons le, est plus fidèle, mais plus difficile à étudier.

5.1.2. Proximité entre cas idéal et cas quelconque

De même, on aimerait bien prolonger les résultats du cas idéal au cas quelconque. Regardons si cela est possible en comparant les deux cas. On présente tout d'abord un graphe en figure 13 (f désigne la fonction qui à tout $n \in [\![2; N]\!]$ associe l'espérance de $T(n)$ dans le cas idéal).

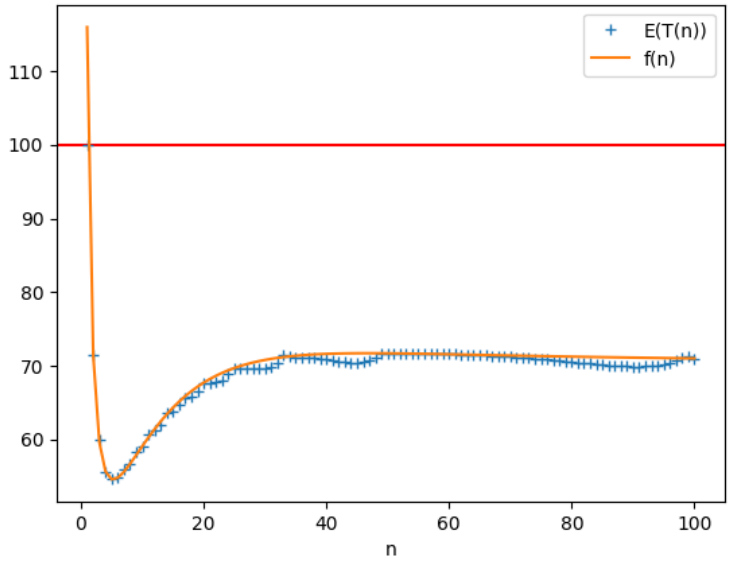


FIGURE 13 – Espérance de $T(n)$ et $f(n)$ en fonction de n pour la division euclidienne, pour $N = 100$, $p = 0{,}1$, $Se = 0{,}7$ et $Sp = 0{,}9$

On voit que l'on a aussi des courbes qui sont très proches. Vérifions comme précédemment cette tendance par une étude statistique.

Code source 5.3 : programme de tracé des courbes 13, et étude statistique des écarts entre les minimums du cas idéal et du cas quelconque, en Python

```
import matplotlib.pyplot as plt
import numpy as np


def f(n, N, p, Se, Sp):
```

```
    return N * (Se + 1/n - (Se+Sp-1)*(1-p)**n)


def espT(n, N, p, Se, Sp):
    if n == 1:
        return N
    q, r = N // n, N % n
    if r == 0:
        return f(n, N, p, Se, Sp)
    if r == 1:
        return q + 1 + q*n*(Se - (Se+Sp-1)*(1-p)**n)
    return N*Se + q + 1 - (Se+Sp-1)*(q*n*(1-p)**n + r*(1-p)**r)


def idealVSqcq(N, p, Se, Sp):
    ns = np.arange(1, N+1)
    es = [espT(n, N, p, Se, Sp) for n in ns]
    fs = [f(n, N, p, Se, Sp) for n in ns]
    plt.clf()
    plt.axhline(N, color = "r")
    plt.plot(ns, es, "+", label = "E(T(n))")
    plt.plot(ns, fs, label = "f(n)")
    plt.xlabel("n")
    plt.legend()
    plt.show()


def min_fonc(N, p, Se, Sp, fonction):
    min_n, min_e = 1, N
    for n in range(2, N+1):
        e = fonction(n, N, p, Se, Sp)
        if e < min_e:
            min_n, min_e = n, e
    return min_n


def statsEcart(N, Se, Sp):
    ps = np.linspace(0.01, 0.99, 100)
    ecarts = [abs(min_fonc(N, p, Se, Sp, espT) - min_fonc(N, p, Se,
    ↪  Sp, f)) for p in ps]
    med = np.median(ecarts)
```

```
    q1, q3 = np.quantile(ecarts, 0.25), np.quantile(ecarts, 0.75)
    d9 = np.quantile(ecarts, 0.9)
    return (q1, med, q3, d9)
```

Voici les résultats de quelques cas :

```
>>> statsEcart(80, 0.7, 0.9)
(0.0, 0.0, 0.0, 0.0)
>>> statsEcart(80, 0.6, 0.7)
(0.0, 0.0, 0.0, 0.0)
>>> statsEcart(100, 0.7, 0.9)
(0.0, 0.0, 0.0, 0.0)
>>> statsEcart(200, 0.9, 0.8)
(0.0, 0.0, 0.0, 0.0)
>>> statsEcart(200, 0.6, 0.5)
(0.0, 0.0, 0.0, 0.0)
>>> statsEcart(1000, 0.65, 0.75)
(0.0, 0.0, 0.0, 0.0)
```

La plupart des minimums sont égaux (on obtient même de meilleurs résultats que dans la précédente comparaison). Pour étudier $\mathbb{E}(T(n))$, on va donc dans la suite seulement regarder la fonction f (cas idéal).

Remarque. Pour l'instant, on n'a pas parlé de seuil comme dans le cas sans bruit. En fait, la spécificité Se a tendance à diminuer l'espérance comme nous allons le voir dans la suite. Il est difficile d'établir des seuils informatiquement, du fait de l'ajout de deux variables par rapport à avant (Se et Sp) : il y a beaucoup de cas à regarder. Cependant, l'étude qui suit va nous permettre de trouver certains seuils.

5.1.3. Étude théorique du cas idéal

Rappelons que f est la fonction définie par :

$$\forall x \in \mathbb{R}_+^*,\ f(x) = N\left(Se + \frac{1}{x} - (Se + Sp - 1)(1-p)^x\right).$$

Ainsi, f est dérivable sur $]0;+\infty[$, et :

$$\forall x > 0,\ f'(x) = \frac{N}{x} - N(Se + Sp - 1)\ln(1-p)\cdot(1-p)^x.$$

Résolvons l'inéquation sur $\mathbb{R}_+^*$:

$$\begin{aligned} f'(x) \geqslant 0 &\iff -\frac{N}{x^2} \geqslant N(Se + Sp - 1)\ln(1-p)\mathrm{e}^{x\ln(1-p)} \\ &\iff (Se + Sp - 1)x^2\ln(1-p)\mathrm{e}^{x\ln(1-p)} \leqslant -1 \end{aligned}$$

$$f'(x) \geqslant 0 \iff (1 - Se - Sp)z(x\ln(1-p)) \leqslant \ln(1-p).$$

On a une première disjonction de cas à faire sur le signe de $1 - Se - Sp$ (qu'on n'était pas amené à faire dans le cas sans bruit car ce nombre vallait -1). En pratique, les valeurs de sensibilité et de spécificité sont telles que $Se + Sp - 1 > 0$, donc $1 - Se - Sp < 0$ (lorsque ce nombre est positif, cela veut dire que les tests sont inefficaces, rendant la procédure peu fiable et inutile). C'est ce qu'on va supposer dans la suite.

Remarque. Si on veut vraiment savoir ce qui se passe lorsque $Se + Sp - 1 \leqslant 0$, il suffit de remarquer qu'alors f est décroissante comme somme de fonctions décroissantes. Le minimum est ainsi en $n = N$.

D'où :

$$f'(x) \geqslant 0 \iff z(x\ln(1-p)) \geqslant \frac{\ln(1-p)}{1 - Se - Sp}.$$

L'argument de z est négatif ; on raisonne dans $\mathbb{R}_-$. Cherchons alors la position relative de $\frac{\ln(1-p)}{1-Se-Sp}$ par rapport au maximum de z sur $]-\infty; 0[$, qui est $4\mathrm{e}^2$.
On remarque que :

$$\frac{\ln(1-p)}{1-Se-Sp} > 4\mathrm{e}^2 \iff \ln(1-p) < 4(1-Se-Sp)\mathrm{e}^2 \iff p > 1 - \mathrm{e}^{4(1-Se-Sp)\mathrm{e}^2}.$$

On a trouvé un premier seuil, dépendant cette fois-ci de Se et Sp.

Notons $\boxed{s_z(Se, Sp) = 1 - \mathrm{e}^{4(1-Se-Sp)\mathrm{e}^2}}$. Si p vérifie $p > s_z(Se, Sp)$, alors f' est négative, donc le minimum est à prendre en $n = N$. On va alors supposer par la suite que $p \leqslant s_z(Se, Sp)$.
Dès lors, nous avons :

$$\begin{aligned} f'(x) \geqslant 0 &\iff Z_-\left(\frac{\ln(1-p)}{1-Se-Sp}\right) \leqslant x\ln(1-p) \leqslant Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right) \\ &\iff \frac{Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right)}{\ln(1-p)} \leqslant x \leqslant \frac{Z_-\left(\frac{\ln(1-p)}{1-Se-Sp}\right)}{\ln(1-p)}. \end{aligned}$$

Posons :

$$\boxed{n_0(p, Se, Sp) = \frac{Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right)}{\ln(1-p)}} \quad \text{et} \quad \boxed{n_-(p, Se, Sp) = \frac{Z_-\left(\frac{\ln(1-p)}{1-Se-Sp}\right)}{\ln(1-p)}}.$$

On en déduit les variations de f sur $\mathbb{R}_+^*$:

x	0	$n_0(p,Se,Sp)$	$n_-(p,Se,Sp)$	$+\infty$
$f'(x)$	$-$	0	$+$ 0	$-$
f	$+\infty$ ↘	$f(n_0(p,Se,Sp))$ ↗	$f(n_-(p,Se,Sp))$ ↘	NSe

Remarque. C'est exactement le même tableau de variation que dans le cas idéal, mais avec des extrema dépendants en plus de Se et Sp, et une limite en $+\infty$ différente.

Étudions maintenant n_0 et n_-.

Les valeurs Se et Sp sont fixes et on s'intéresse à des seuils concernant p. Ainsi, on notera plutôt $n_0(p)$ et $n_-(p)$ pour simplifier.

En reprenant le calcul de la dérivée par rapport à p dans le cas sans bruit, on trouve pour p différent de $s_z(Se,Sp)$:

$$n_0'(p) = \frac{Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right)}{(1-p)\ln^2(1-p)}\left(1 - \frac{1}{Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right)+2}\right)$$

et de même pour $n_-'(p)$ en replaçant 0 par $-$.

On sait que $-2 \leqslant Z_0 \leqslant 0$ donc pour $p \in]0, s_z(Se,Sp)[$, on a :

$$\begin{aligned}
n_0'(p) \geqslant 0 &\iff 1 - \frac{1}{Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right)+2} \leqslant 0 \\
&\iff \frac{1}{Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right)+2} \geqslant 1 \\
&\iff Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right)+2 \leqslant 1 \\
&\iff Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right) \leqslant -1 \\
&\iff \frac{\ln(1-p)}{1-Se-Sp} \geqslant z(-1) = \mathrm{e}^{-1} \\
&\iff \ln(1-p) \leqslant (1-Se-Sp)\mathrm{e}^{-1} \\
n_0'(p) \geqslant 0 &\iff p \geqslant 1 - \mathrm{e}^{(1-Se-Sp)\mathrm{e}^{-1}}.
\end{aligned}$$

Posons $\boxed{p_0(Se,Sp) = 1 - \mathrm{e}^{(1-Se-Sp)\mathrm{e}^{-1}}}$. Comme $\mathrm{e}^{-1} \approx 0{,}37$ et $4\mathrm{e}^{-2} \approx 0{,}54$, on a $\mathrm{e}^{-1} < 4\mathrm{e}^{-2}$, donc $(1-Se-Sp)\mathrm{e}^{-1} > 4(1-Se-Sp)\mathrm{e}^{-2}$, et donc $p_0(Se,Sp) < s_z(Se,Sp)$.

Calculons :

$$n_0(p_0(Se,Sp)) = \frac{Z_0(\mathrm{e}^{-1})}{(1-Se-Sp)\mathrm{e}^{-1}} = \frac{-1}{(1-Se-Sp)\mathrm{e}^{-1}} = \frac{\mathrm{e}}{Se+Sp-1}.$$

Et :

$$n_0(s_z(Se,Sp)) = \frac{Z_0(4\mathrm{e}^{-2})}{4(1-Se-Sp)\mathrm{e}^{-2}} = \frac{-2}{4(1-Se-Sp)\mathrm{e}^{-2}} = \frac{\mathrm{e}^2}{2(Se+Sp-1)}.$$

Enfin, au voisinage de 0, on a :

$$\begin{aligned} n_0(p) &= \frac{Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right)}{(1-Se-Sp){Z_0}^2\left(\frac{\ln(1-p)}{1-Se-Sp}\right)\exp\left(Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right)\right)} \\ &= \frac{1}{(1-Se-Sp)Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right)\exp\left(Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right)\right)}. \end{aligned}$$

Or, $\frac{\ln(1-p)}{1-Se-Sp} \xrightarrow[p\to 0]{} 0^+$ et par continuité de Z_0 en 0, on a $Z_0(X) \xrightarrow[X\to 0^+]{} 0^-$. D'où $n_0(p) \xrightarrow[p\to 0]{} +\infty$.

On en déduit les variations de n_0 :

p	0		$p_0(Se,Sp)$		$s_z(Se,Sp)$
$n_0'(p)$	‖	$-$	0	$+$	‖
n_0	‖ $+\infty$	↘	$\frac{\mathrm{e}}{Se+Sp-1}$	↗	$\frac{\mathrm{e}^2}{2(Se+Sp-1)}$

Or, $Se, Sp \in [0,1]$ donc $Se+Sp \leqslant 2$ donc $\frac{1}{Se+Sp-1} \geqslant 1$. Ainsi, $n_0 \geqslant \frac{\mathrm{e}}{Se+Sp-1} \geqslant \mathrm{e} > 2$.
Cherchons maintenant $p \in]0; s_z(Se,Sp)[$ tel que $n_0(p) < N$. On a :

$$n_0(p) < N \iff Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right) > N\ln(1-p).$$

Or, on a :

$$N\ln(1-p) < -2 \iff 1-p < \mathrm{e}^{-\frac{2}{N}} \iff p > 1-\mathrm{e}^{-\frac{2}{N}}.$$

Posons $\boxed{s(N) = 1-\mathrm{e}^{-\frac{2}{N}}}$. Comme N est assez grand, on peut supposer $N \geqslant \frac{\mathrm{e}^2}{2(Se+Sp-1)}$ (les valeurs de Se et Sp sont généralement telles que $Se+Sp-1 \gg 0$ donc la fraction n'est pas très grande). Ainsi, on a :

$$\frac{1}{N} \leqslant \frac{2(Se+Sp-1)}{\mathrm{e}^2} \qquad \text{donc} \qquad -\frac{2}{N} \geqslant 4(1-Se-Sp)\mathrm{e}^{-2}.$$

D'où $s(N) \leqslant s_z(Se,Sp)$.

On en déduit que l'ensemble $]s(N); s_z(Se, Sp)[$ est solution de l'équation $n_0(p) < N$. Regardons alors dans $]0; s(N)]$ On a :

$$\begin{aligned} n_0(p) < N &\iff \frac{\ln(1-p)}{1-Se-Sp} < z(N\ln(1-p)) \\ &\iff \frac{\ln(1-p)}{1-Se-Sp} < N^2\ln^2(1-p)\,\mathrm{e}^{N\ln(1-p)} \\ &\iff \frac{1}{1-Se-Sp} > N^2\ln(1-p)\,\mathrm{e}^{N\ln(1-p)} \\ n_0(p) < N &\iff w(N\ln(1-p)) < \underbrace{\frac{1}{N(1-Se-Sp)}}_{<0}. \end{aligned}$$

Or, $N \geqslant \frac{\mathrm{e}^2}{2(Se+Sp-1)} > \frac{\mathrm{e}}{Se+Sp-1}$. Donc :

$$\frac{1}{N} < (Se+Sp-1)\mathrm{e}^{-1} \qquad \text{donc} \qquad \frac{1}{N(1-Se-Sp)} > -\mathrm{e}^{-1}.$$

Ainsi :

$$\begin{aligned} n_0(p) < N &\iff W_{-1}\left(\frac{1}{N(1-Se-Sp)}\right) \leqslant N\ln(1-p) \leqslant W_0\left(\frac{1}{N(1-Se-Sp)}\right) \\ &\iff \frac{1}{N}W_{-1}\left(\frac{1}{N(1-Se-Sp)}\right) \leqslant \ln(1-p) \leqslant \frac{1}{N}W_0\left(\frac{1}{N(1-Se-Sp)}\right) \\ n_0(p) < N &\iff 1-\mathrm{e}^{\frac{1}{N}W_0\left(\frac{1}{N(1-Se-Sp)}\right)} \leqslant p \leqslant 1-\mathrm{e}^{\frac{1}{N}W_{-1}\left(\frac{1}{N(1-Se-Sp)}\right)} \end{aligned}$$

Posons :

$$\boxed{s_0(N,Se,Sp) = 1-\exp\left(\frac{1}{N}W_0\left(\frac{1}{N(1-Se-Sp)}\right)\right)}$$

$$\boxed{s_{-1}(N,Se,Sp) = 1-\exp\left(\frac{1}{N}W_{-1}\left(\frac{1}{N(1-Se-Sp)}\right)\right)}.$$

Étudions leur position relative par rapport à $s(N)$.
On a $W_0 > -2$ donc :

$$W_0\left(\frac{1}{N(1-Se-Sp)}\right) > -2 \qquad \text{donc} \qquad \frac{1}{N}W_0\left(\frac{1}{N(1-Se-Sp)}\right) > -\frac{2}{N}.$$

D'où $s_0(N,Se,Sp) < s(N)$ (et $s_0(N,Se,Se) < s_0(N,Se,Sp)$).
Par ailleurs, on a $N \geqslant \frac{\mathrm{e}^2}{2(Se+Sp-1)}$ donc :

$$\frac{1}{N} \leqslant 2(Se+Sp-1)\mathrm{e}^{-2} \qquad \text{donc} \qquad \frac{1}{N(1-Se-Sp)} \geqslant -2\mathrm{e}^{-2} = w(-2).$$

Ainsi, on en déduit :

$$W_{-1}\left(\frac{1}{N(1-Se-Sp)}\right) \leqslant -2 \qquad \text{donc} \qquad \frac{1}{N}W_{-1}\left(\frac{1}{N(1-Se-Sp)}\right) \leqslant -\frac{2}{N}.$$

D'où $s_{-1}(N, Se, Sp) \geqslant s(N)$.

Pour conclure, dans $]0; s(N)]$, on a :

$$n_0(p) < N \iff s_0(N, Se, Sp) < p \leqslant s(N) \iff p > s_0(N, Se, Sp).$$

Et dans $]0; s_z(Se, Sp)[$, on a :

$$n_0(p) < N \iff s_0(N, Se, Sp) < p < s_z(Se, Sp) \iff p > s_0(N, Se, Sp).$$

Ainsi, lorsque $p \leqslant s_0(N, Se, Sp)$, $n_0(p)$ est au delà de N. Donc f décroît sur $[1; N]$ et donc la valeur qui minimise l'espérance est $n = N$. Il nous reste à étudier n_- dans le cas où p est dans $]s_0(N, Se, Sp); s_z(Se, Sp)]$.

Au préalable, on a $n_- \geqslant n_0$ donc $n_- > 2$. On a pas besoin d'étudier les variations de n_- car on a déjà la minoration qu'on veut.

Cherchons maintenant p dans l'intervalle ci-dessus tel que $n_-(p) < N$. On a :

$$n_-(p) < N \iff Z_-\left(\frac{\ln(1-p)}{1-Se-Sp}\right) > N\ln(1-p)$$

Dans $]s_0(N, Se, Sp), s(N)[$, on a $N\ln(1-p) > -2$ donc nécessairement, $n_-(p) \leqslant N$.

Plaçons nous alors dans $[s(N); s_z(Se, Sp)]$. Ainsi :

$$\begin{aligned}
n_-(p) < N &\iff \frac{\ln(1-p)}{1-Se-Sp} > z(N\ln(1-p)) \\
&\iff \frac{\ln(1-p)}{1-Se-Sp} > N^2\ln^2(1-p)\,\mathrm{e}^{N\ln(1-p)} \\
&\iff w(N\ln(1-p)) > \frac{1}{N(1-Se-Sp)} \\
&\iff N\ln(1-p) < W_{-1}\left(\frac{1}{N(1-Se-Sp)}\right) \\
&\quad \text{ou} \qquad N\ln(1-p) > W_0\left(\frac{1}{N(1-Se-Sp)}\right) \\
&\iff p < s_0(N, Se, Sp) \qquad \text{ou} \qquad p > s_{-1}(N, Se, Sp) \\
n_-(p) < N &\iff p > s_{-1}(N, Se, Sp).
\end{aligned}$$

Il nous faut étudier la position de $s_{-1}(N, Se, Sp)$ par rapport à $s_z(Se, Sp)$.

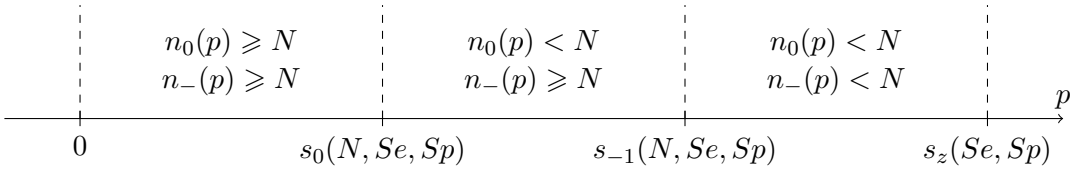
On admet que $s_{-1}(N, Se, Sp) \leqslant s_z(Se, Sp)$ (pour le montrer, il faut prouver qu'une certaine fonction compliquée est positive ; on le voit bien graphiquement).

Et on a déja vu que $s(N) \leqslant s_{-1}(N, Se, Sp)$.

On conclut que dans $]s_0(N, Se, Sp); s_z(Se, Sp)]$, on a :

$$n_-(p) < N \iff s_{-1}(N, Se, Sp) < p \leqslant s_z(Se, Sp) \iff s_{-1}(N, Se, Sp) < p.$$

Résumons les différents cas. Pour N assez grand (supérieur ou égal à $\frac{\mathrm{e}^2}{2(Se+Sp-1)}$) :



On en déduit la position du minimum en fonction des cas :
— pour $p \leqslant s_0(N, Se, Sp)$, le minimum de f est en N ;
— pour $s_0(N, Se, Sp) < p \leqslant s_{-1}(N, Se, Sp)$, le minimum est en $n_0(p, Se, Sp)$;
— pour $s_{-1}(N, Se, Sp) < p \leqslant s_z(Se, Sp)$, le minimum de f est soit en $n_0(p, Se, Sp)$, soit en N.
Dans le dernier cas, on ne peut pas conclure (la théorie devient trop compliquée).

5.1.4. Résumé

Avec la proximité entre cas idéal et cas quelconque, et avec la proximité des deux méthodes, on en conclut le théorème suivant :

Théorème 5.1 : détermination de la conception optimisant $\mathbb{E}(T(n))$ avec des groupes de même taille (à 1 près) pour une population homogène, avec un test non parfait

Pour une population homogène assez grande ($N \geqslant \frac{\mathrm{e}^2}{2(Se+Sp-1)}$), pour obtenir des groupes de même taille (à 1 près) optimisant le nombre de tests dans la procédure du test groupé, où le test n'est pas parfait :
— si $p \leqslant 1 - \mathrm{e}^{\frac{1}{N} W_0\left(\frac{1}{N(Se+Sp-1)}\right)}$, on prend $\tau = (N)$, *i.e.* on prend l'entière population comme unique groupe ;
— si $1 - \mathrm{e}^{\frac{1}{N} W_0\left(\frac{1}{N(Se+Sp-1)}\right)} < p \leqslant 1 - \mathrm{e}^{\frac{1}{N} W_{-1}\left(\frac{1}{N(Se+Sp-1)}\right)}$, on calcule $\frac{1}{\ln(1-p)} Z_0\left(\frac{\ln(1-p)}{1-Se-Sp}\right)$, on prend l'entier n le plus proche de ce nombre, puis on calcule la division euclidienne de N par n. Ensuite, en notant le résultat $N = nq + r$:
  — si $r = 0$, on prend $\tau = (\underbrace{n, \ldots, n}_{q \text{ fois}})$;
  — sinon, on calcule $\textsc{CombCons}(n, q, r) = (\alpha, u, v)$, puis on prend :

$$\tau = (\underbrace{\alpha, \ldots, \alpha}_{u \text{ fois}}, \underbrace{\alpha+1, \ldots, \alpha+1}_{v \text{ fois}});$$

— si $1 - \mathrm{e}^{\frac{1}{N} W_{-1}\left(\frac{1}{N(Se+Sp-1)}\right)} < p \leqslant 1 - \mathrm{e}^{4(1-Se-Sp)\mathrm{e}^{-2}}$, on est dans une zone incertaine : on doit calculer et comparer les deux conceptions obtenues dans les deux cas précédents ;
— si $p > 1 - \mathrm{e}^{4(1-Se-Sp)\mathrm{e}^{2}}$, on prend $\tau = (N)$.

Remarque. Visiblement, on n'a pas de seuil au delà duquel la procédure n'est pas optimale, comme on avait trouvé dans le cas sans bruit. En fait, la limite en $+\infty$ est NSe et non N. Il est difficile d'étudier l'équation $f(x) = N$ (mais facile d'étudier $f(x) = NSe$!). On est obligé de calculer la conception optimale comme ci-dessus pour ensuite conclure sur l'optimalité de la procédure.

5.2. Problème de la spécificité et nouveau critère

Comme on l'a vu avec l'étude de f, la fonction tend vers NSe en $+\infty$: c'est une valeur approchée de l'espérance minimale dans certains cas (minimum en $n = N$ avec N grand). *A priori*, on serait donc amené à diminuer le plus possible la sensibilité pour optimiser l'espérance du nombre de test. C'est absurde car cela rend la procédure de moins en moins fiable.

En fait, notre critère d'optimalité n'est pas le plus pertinent. Et ce n'est pas celui que les auteurs prennent dans le cas du test non parfait. Avant de le présenter, nous devons introduire deux nouvelles variables aléatoires.

5.2.1. Nombre de faux positifs et nombre de faux négatifs

Définition 5.1

Pour une conception τ, on note $FP(\tau)$ la variable aléatoire discrète qui associe le nombre de faux positifs à l'issue du processus du test groupé.
On définit de même $FN(\tau)$ pour le nombre de faux négatifs.

Ces variables aléatoires sont bien sûr nulles si le test est parfait (c'est pourquoi on ne les a pas utilisées précédemment).

L'autre critère d'optimalité utilisé est alors le suivant [3] : on fixe $\lambda \in [0;1]$ et on cherche à minimiser $\lambda\mathbb{E}(FN(\tau)) + (1-\lambda)\mathbb{E}(FP(\tau))$ tout en veillant à ne pas dépasser une certaine valeur seuil B pour $\mathbb{E}(T(\tau))$.

Remarque. On verra dans la partie pour les populations hétérogènes qu'on impose aussi à la conception optimale d'avoir des tailles de groupe inférieures à une limite donnée $l \geqslant 2$ ($l = 1$ correspond au cas « classique »). Ici, on ne considère pas cet autre seuil car il est en fait presque inutile. En effet, si on prenait en compte l, on aurait deux cas : soit les partitions optimales sont de taille trop faible devant l'entier l imposé, soit elles n'existent pas, ce qui réduit ici l'intérêt de l à un simple seuil d'existence : il ne modifie aucunement la manière dont les groupes sont formés. Si on veut vraiment limiter la taille des groupes, on regardera juste le résultat donné sans ce critère et on conclura en regardant la taille des groupes (qui sont globalement les mêmes ; *cf.* la suite).

5.2.2. Formules de l'espérance du nombre de faux positifs et de faux négatifs

Comme pour $T(n)$, on a aussi des formules pour calculer $FN(n)$ et $FP(n)$. On donne les formules dans le cas général. De même, on les admet (*cf.* démonstrations dans l'article [3]).

Propriété 5.3 : formule de l'espérance du nombre de faux positifs dans le cas homogène

$$\mathbb{E}(FP(\tau)) = (1-Sp)\sum_{\substack{n\in\tau\\ n=1}}(1-p) + (1-Sp)(1-p)\sum_{\substack{n\in\tau\\ n\neq 1}} n\big(Se-(Se+Sp-1)(1-p)^{n-1}\big).$$

Propriété 5.4 : formule de l'espérance du nombre de faux négatifs dans le cas homogène

$$\mathbb{E}(FN(\tau)) = (1-Se)\sum_{\substack{n\in\tau\\ n=1}} p + (1-Se^2)\sum_{\substack{n\in\tau\\ n\neq 1}} np.$$

Comme avant, $\mathbb{E}(FP(n))$ et $\mathbb{E}(FN(n))$ vont dépendre de la décomposition choisie. Avec les deux formules ci-dessus, on peut les calculer :

Propriété 5.5 : formules des espérances du nombre de faux positifs et du nombre de faux positifs pour la méthode de la division euclidienne

Soit $n \in [\![2; N]\!]$. Notons $N = nq + r$ la division euclidienne de N par n. Nous avons :

— si $r = 0$, alors :

$$\mathbb{E}(FP(n)) \overset{\text{DE}}{=} N(1-Sp)(1-p)\left(Se-(Se+Sp-1)(1-p)^{n-1}\right)$$

$$\mathbb{E}(FN(n)) \overset{\text{DE}}{=} N\left(1-Se^2\right)p;$$

— si $r = 1$, alors :

$$\mathbb{E}(FP(n)) \overset{\text{DE}}{=} (1-Sp)(1-p)\left(1+(N-1)\left(Se-(Se+Sp-1)(1-p)^{n-1}\right)\right)$$

$$\mathbb{E}(FN(n)) \overset{\text{DE}}{=} (1-Se)p\left(N(1+Se)-Se\right);$$

— si $r > 1$, alors :

$$\mathbb{E}(FP(n)) \overset{\text{DE}}{=} (1-Sp)(1-p)\left(NSe-(Se+Sp-1)\left(qn(1-p)^{n-1}+r(1-p)^{r-1}\right)\right)$$

$$\boxed{\mathbb{E}(FN(n)) \overset{\text{DE}}{=} N\left(1 - Se^2\right) p}.$$

Propriété 5.6 : formules des espérances du nombre de faux positifs et du nombre de faux positifs pour la méthode de la combinaison de deux entiers consécutifs

Soit $n \in [\![2; N]\!]$. Notons $N = \alpha u + (\alpha + 1)v$ la décomposition de N en somme de deux entiers consécutifs. On a :

— si $\alpha = 0$, alors $N = v$ et il s'agit du cas « classique » :

$$\boxed{\mathbb{E}(FP(n)) \overset{\text{CC}}{=} N(1 - Sp)}$$

$$\boxed{\mathbb{E}(FN(n)) \overset{\text{CC}}{=} N(1 - Se)};$$

— si $\alpha = 1$, alors :

$$\boxed{\mathbb{E}(FP(n)) \overset{\text{CC}}{=} (1 - Sp)(1 - p)\big(u + 2v\left(Se - (Se + Sp - 1)(1 - p)\right)\big)}$$

$$\boxed{\mathbb{E}(FN(n)) \overset{\text{CC}}{=} (1 - Se)p\left(N + 2vSe\right)};$$

— si $\alpha > 1$, alors :

$$\boxed{\begin{aligned}\mathbb{E}(FP(n)) &\overset{\text{CC}}{=} \\ &(1 - Sp)\big(NSe - (Se + Sp - 1)(1 - p)^{\alpha}\left(N - p(\alpha + 1)v\right)\big)\end{aligned}}$$

$$\boxed{\mathbb{E}(FN(n)) \overset{\text{CC}}{=} N\left(1 - Se^2\right) p}.$$

Maintenant qu'on a les formules, on va rechercher la partition respectant le nouveau critère.

5.2.3. Recherche de la conception optimale pour le nouveau critère

Regardons d'abord l'allure de la fonction $n \mapsto \lambda\,\mathbb{E}(FN(n)) + (1 - \lambda)\mathbb{E}(FP(n))$ pour N, p, Se, Sp fixé, et pour les deux décompositions.

On voit avec les figures 14 et 15 que quelque soit λ dans $[0; 1]$, la fonction qui associe la quantité que l'on cherche à minorer est toujours croissante. On observe le résultat pour d'autres valeurs des paramètres N, p, Se et Sp.

On peut aussi comparer les deux décompositions : d'après la figure 16, les variations sont les mêmes pour les deux décompositions.

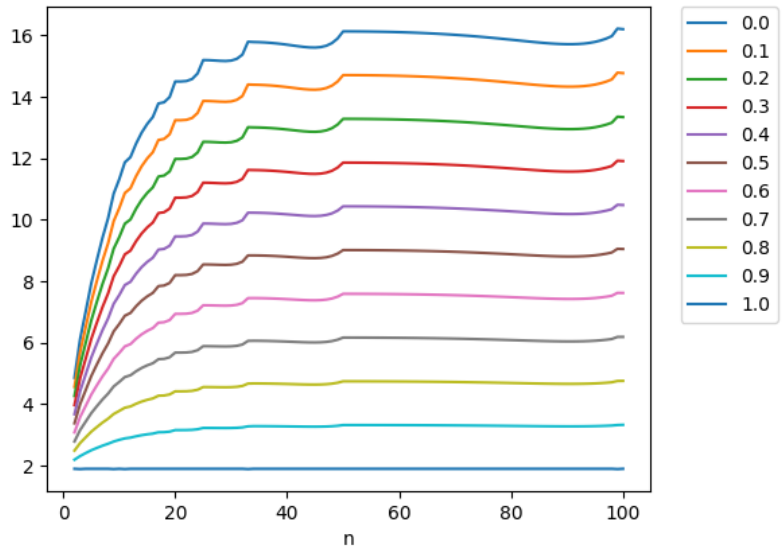


FIGURE 14 – $\lambda\,\mathbb{E}(FN(n)) + (1-\lambda)\mathbb{E}(FP(n))$ en fonction de n pour la méthode de la division euclidienne, pour $N = 100$, $p = 0{,}1$, $Se = 0{,}9$ et $Sp = 0{,}8$

Voici le programme de tracé des courbes présentées :

Code source 5.4 : programme de tracé des courbes de la combinaison barycentrique des espérances du nombre de faux positifs et de faux négatifs, en Python

```
import numpy as np
import matplotlib.pyplot as plt


def espFP_DE(n, N, p, Se, Sp):
    if n == 1:
        return N * (1 - Sp)
    q, r = N // n, N % n
    if r == 0:
        return N*(1-Sp)*( Se*(1-p) - (Se+Sp-1)*(1-p)**n )
    if r == 1:
        return (1-Sp) * ( (1-Se)*(1-p) + N*Se*(1-p) -
        ↪  (N-1)*(Se+Sp-1)*(1-p)**n )
    return (1-Sp)*( N*Se*(1-p) - (Se+Sp-1)*(q*n*(1-p)**n +
    ↪  r*(1-p)**r))
```

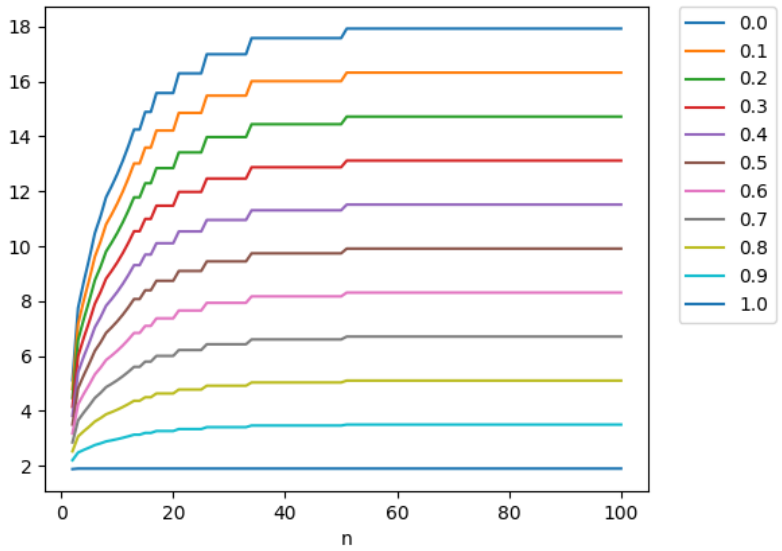


FIGURE 15 – $\lambda\,\mathbb{E}(FN(n)) + (1-\lambda)\mathbb{E}(FP(n))$ en fonction de n pour la méthode de la combinaison consécutive, pour $N = 100$, $p = 0{,}1$, $Se = 0{,}9$ et $Sp = 0{,}8$

```python
def espFN_DE(n, N, p, Se, Sp):
    if n == 1:
        return N * (1 - Se)
    q, r = N // n, N % n
    if r == 0:
        return N*(1-Se**2)*p
    if r == 1:
        return (1-Se)*p*(N*(Se+1) - Se)
    return N * (1-Se**2) * p


def comb_cons(n, q, r):
    assert r < n
    while n - r > q:
        n -= 1
        r += q
    return (n-1, n-r, q+r-n+1)
```

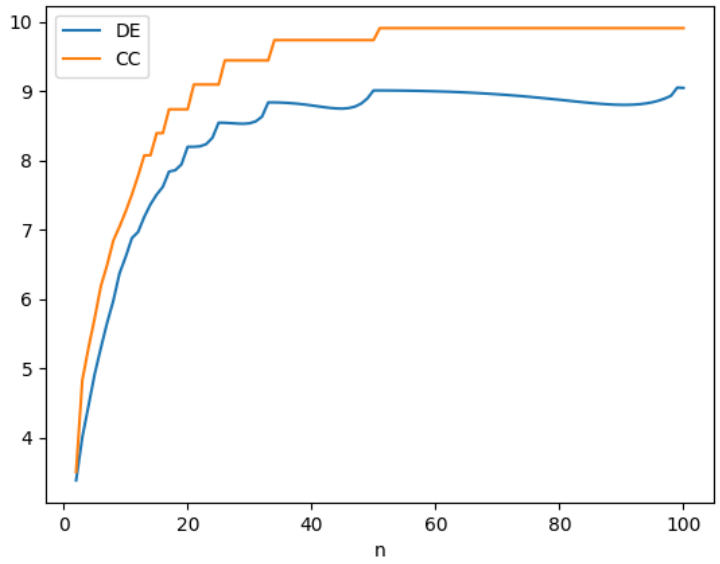


FIGURE 16 – $\lambda\,\mathbb{E}(FN(n)) + (1-\lambda)\mathbb{E}(FP(n))$ en fonction de n pour la méthode de la combinaison consécutive, pour $N = 100$, $p = 0{,}1$, $Se = 0{,}9$ et $Sp = 0{,}8$

```
def espFP_CC(n, N, p, Se, Sp):
    if n == 1:
        return N * (1 - Sp)
    q, r = N // n, N % n
    (alpha, u, v) = comb_cons(n, q, r)
    if alpha == 0:
        return N * (1 - Sp)
    if alpha == 1:
        return (1-Sp) * ( (1-p)*(u+2*v*Se) -
        ↪  2*v*(Se+Sp-1)*(1-p)**2)
    return (1-Sp) * (N*Se -
    ↪  (Se+Sp-1)*(N-p*(alpha+1)*v)*(1-p)**alpha)


def espFN_CC(n, N, p, Se, Sp):
    if n == 1:
        return N * (1 - Se)
```

```
    q, r = N // n, N % n
    (alpha, u, v) = comb_cons(n, q, r)
    if alpha == 0:
        return N * (1 - Se)
    if alpha == 1:
        return (1-Se) * p * (N+2*Se*v)
    return N * (1-Se**2) * p


def courbeFNP_DE(N, p, Se, Sp):
    ns = np.arange(2, N+1)
    lamb = [ k / 10 for k in range(0, 11)]
    fig = plt.figure()
    ax = fig.add_axes([0.13, 0.15, 0.7, 0.75])
    for l in lamb :
        comb = [l * espFN_DE(n, N, p, Se, Sp) + (1-l) * espFP_DE(n,
        ↪  N, p, Se, Sp) for n in ns]
        ax.plot(ns, comb, label = str(round(l, 1)))
    ax.set_xlabel("n")
    ax.legend(bbox_to_anchor=(1.05, 1), loc='upper left',
    ↪  borderaxespad=0.)
    plt.show()


def courbeFNP_CC(N, p, Se, Sp):
    ns = np.arange(2, N+1)
    lamb = [ k / 10 for k in range(0, 11)]
    fig = plt.figure()
    ax = fig.add_axes([0.13, 0.15, 0.7, 0.75])
    for l in lamb :
        comb = [l * espFN_CC(n, N, p, Se, Sp) + (1-l) * espFP_CC(n,
        ↪  N, p, Se, Sp) for n in ns]
        ax.plot(ns, comb, label = str(round(l, 1)))
    ax.set_xlabel("n")
    ax.legend(bbox_to_anchor=(1.05, 1), loc='upper left',
    ↪  borderaxespad=0.)
    plt.show()


def comparerDecompositions(N, p, Se, Sp, lamb):
    ns = np.arange(2, N+1)
```

```
    c1 = [lamb * espFN_DE(n, N, p, Se, Sp) + (1-lamb) * espFP_DE(n,
    ↪  N, p, Se, Sp) for n in ns]
    c2 = [lamb * espFN_CC(n, N, p, Se, Sp) + (1-lamb) * espFP_CC(n,
    ↪  N, p, Se, Sp) for n in ns]
    plt.figure()
    plt.plot(ns, c1, label = "DE")
    plt.plot(ns, c2, label = "CC")
    plt.xlabel("n")
    plt.legend()
    plt.show()


courbeFNP_DE(100, 0.1, 0.9, 0.8)
courbeFNP_CC(100, 0.1, 0.9, 0.8)
comparerDecompositions(100, 0.1, 0.9, 0.8, 0.5)
```

Dans tous les cas, on a donc intérêt à prendre n le plus petit possible. Mais on rappelle que l'on veut aussi contrôler en parallèle le nombre de tests utilisés, avec la condition $\mathbb{E}(T(n)) \leqslant B$. On en déduit le théorème final :

Théorème 5.2 : détermination pour le nouveau critère de la conception avec des groupes de même taille (à 1 près) pour une population homogène, avec un test non parfait

Pour une population homogène, pour obtenir des groupes de même taille (à 1 près) optimisant la fiabilité la procédure du test groupé et utilisant un nombre maximal B de tests en moyenne, où le test n'est pas parfait, on choisit le plus petit $n \in [\![2; N]\!]$ tel que $E(T(n)) \leqslant B$, sous réserve d'existence, on calcule la décomposition en somme de deux entiers consécutifs puis on en déduit la partition.

Le théorème 5.1 peut aider à trouver n vérifiant $E(T(n)) \leqslant B$ mais il ne donne pas forcément l'entier optimal. Son aide est d'antant plus utile que B est faible. En effet, plus B diminue, moins il y a d'entiers n qui conviennent. Mais celui optimisant $E(T(n))$ sera toujours présent parmi ces entiers, jusqu'à qu'il n'y en ait plus.

Maintenant qu'on a pleinement étudié le test non parfait avec des groupes de même taille (à 1 près), on va regarder les groupes de taille quelconque.

5.3. Groupes de taille quelconque

On considère maintenant toutes les partitions de N. On va tout de suite considérer le nouveau critère introduit dans la section précédente car l'essence même du cas non parfait est de maximiser la fiabilité plus que le nombre de tests.

5.3.1. Parcours de l'ensemble des partitions

On a déjà écrit un programme calculant l'ensemble des partitions de N. Il nous reste juste à modifier la fonction `espT` précédente en lui ajoutant deux paramètres `se` et `sp`. Il nous faut aussi écrire deux fonctions qui à une partition associent respectivement l'espérance du nombre de faux positifs et l'espérance du nombre de faux négatifs.

Code source 5.5 : détermination de la conception optimale selon le deuxième critère, si elle existe, pour des groupes de taille quelconque, en OCaml

```
(* calcul de la liste des partitions *)
let scinde l =
  let rec compter1 l nb =
    match l with
    | 1 :: t -> compter1 t (nb+1)
    | _ -> (nb, l)
  in
  compter1 l 0 ;;


let rec ajoute n x l =
  match n with
  | 0 -> l
  | n -> x :: (ajoute (n-1) x l) ;;


let partition_suivante l =
  let (n, u) = scinde l in
  match u with
  | [] -> []
  | t :: v -> begin
    let q = (n + t) / (t - 1)
    and r = (n + t) mod (t - 1) in
    match r with
    | 0 -> ajoute q (t-1) v
    | _ -> r :: (ajoute q (t-1) v)
  end ;;


let partitions n =
  let rec cons l =
    let c = partition_suivante (List.hd l) in
      match c with
```

```
      | [] -> l
      | _ -> cons (c::l)
  in
  cons [[n]] ;;


(* calcul des espérances *)
let espT part p se sp =
  let q = List.length part in
  let rec somme l = match l with
    | [] -> Float.of_int q
    | a :: t when a > 1 -> (
      let n = Float.of_int a in
      let s = n *. (se -. (se +. sp -. 1.) *. (1. -. p) ** n) in
      s +. (somme t)
     )
    | _ :: t -> somme t
  in
  somme part ;;


let espFP part p se sp =
  let (nb1, partSans1) = scinde part in
  let nb1_f = Float.of_int nb1 in
  let somme1 = (1. -. sp) *. nb1_f *. (1. -. p) in
  let rec calculSommeSans1 l =
    match l with
    | [] -> 0.
    | n :: t -> (
      let n_f = Float.of_int n in
      let s = n_f *. (se -. (se +. sp -. 1.) *. (1. -. p) ** (n_f
      ↪  -. 1.) ) in
      s +. calculSommeSans1 t
    )
  in
  let sommeSans1 = (1. -. sp) *. (1. -. p) *. calculSommeSans1
  ↪  partSans1 in
  somme1 +. sommeSans1 ;;


let espFN part p se sp =
```

```
  let (nb1, partSans1) = scinde part in
  let nb1_f = Float.of_int nb1 in
  let somme1 = (1. -. se) *. nb1_f *. p in
  let rec calculSommeSans1 l =
    match l with
    | [] -> 0.
    | n :: t -> (
      let n_f = Float.of_int n in
      n_f +. calculSommeSans1 t
    )
  in
  let sommeSans1 = (1. -. se ** 2.) *. p *. calculSommeSans1
  ↪  partSans1 in
  somme1 +. sommeSans1 ;;


(* calcul de la conception optimale *)
let conceptionOpt nPop p se sp lambda b =
  let listePart = partitions nPop in
  let rec rechercheMin l partAct minAct trouve =
    match l with
    | [] when trouve -> (
        let e = espT partAct p se sp in
        (partAct, minAct, e)
      )
    | [] -> failwith "La partition optimale pour des groupes
    ↪  quelconques n'existe pas"
    | part :: t -> let et = espT part p se sp in
      if et > b then rechercheMin t partAct minAct trouve
      else (
        let efp = espFP part p se sp
        and efn = espFN part p se sp in
        let c = lambda *. efn +. (1. -. lambda) *. efp in
        if c < minAct then
          rechercheMin t part c true
        else rechercheMin t partAct minAct trouve
      )
  in
  rechercheMin listePart [] max_float false ;;
```

Voici quelques résultats (on rappelle que notre recherche se limite aux tailles de population inférieures à 80 du fait de la lenteur du calcul des partitions) :

```
# conceptionOpt 60 0.1 0.9 0.8 0.5 60. ;;
- : int list * float * float =
([2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2;
  2; 2; 2; 2; 2],
 2.0279999999999978, 49.979999999999187)
# conceptionOpt 60 0.1 0.9 0.8 0.5 45. ;;
- : int list * float * float =
([2; 2; 2; 2; 2; 2; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3],
 2.30015999999999821, 44.7015999999999849)
# conceptionOpt 60 0.1 0.9 0.8 0.5 40. ;;
Exception:
Failure "La partition optimale pour des groupes quelconques n'existe pas".
```

Pour optimiser notre recherche de la meilleure partition, on va comme avant essayer d'établir des propriétés sur les meilleures conceptions.

5.3.2. Propriétés des partitions optimales

Remarque. On ne va pas faire de recherche de seuil d'optimalité car on a maintenant 4 nouveaux paramètres à gérer, qui sont Se, Sp, λ et B. Il est difficile de dégager un seuil général. On admettra qu'il existe bel et bien un seuil, mais propre à Se, Sp, λ et B.

On va d'abord regarder la fréquence de partitions optimales qui sont des combinaisons consécutives.

On écrit une fonction `frequenceCC : int -> float -> float -> float -> float -> float` qui prend en paramètre N, Se, Sp, λ et B et qui pour 100 valeurs de p réparties uniformément dans $]0;1[$ renvoie la fréquence des conceptions qui sont des combinaisons de deux entiers consécutifs.

Code source 5.6 : calcul de la fréquence de partitions optimales qui sont des combinaisons consécutives, en OCaml

```
(* calcul de la fréquence de CC parmi les conceptions optimales *)
let estCC l =
  let rec listeCste l =
    match l with
    | [] -> true
    | [a] -> true
    | a :: b :: t when a = b -> listeCste (b :: t)
    | _ -> false
  in
  let rec chercherDeuxieme l =
    match l with
    | [] -> true
    | [a] -> true
```

```
    | a :: b :: t when a = b -> chercherDeuxieme (b :: t)
    | a :: b :: t when b = a + 1 -> listeCste (b :: t)
    | _ -> false
  in
  chercherDeuxieme l ;;


(* récupère le troisième élément d'un triplet *)
let trd_tr (a, b, c) = c ;;


(* fréquence des combinaisons consécutives *)
let frequenceCC nPop se sp lambda b =
  let rec comptage nb eff ps =
    match ps with
    | [] -> (nb, eff)
    | p :: t -> try (
      let popt = trd_tr (conceptionOpt nPop p se sp lambda b) in
        if estCC popt then comptage (nb + 1) (eff + 1) t
        else comptage nb (eff + 1) t
      )
      with
        | _ -> comptage nb eff t
  in
  let f i =
    let i_f = Float.of_int i in (i_f +. 1.) /. 101.
  in
  let (nb, eff) = List.init 100 f |> (comptage 0 0) in
  try
    (Float.of_int nb) /. (Float.of_int eff)
  with
    | Division_by_zero -> failwith "Rien n'a été compté" ;;
```

Voici quelques résultats :

```
# frequenceCC 40 0.9 0.8 0.5 30. ;;
- : float = 1.
# frequenceCC 40 0.9 0.8 0.5 36. ;;
- : float = 1.
# frequenceCC 40 0.9 0.8 0.5 37. ;;
- : float = 1.
# frequenceCC 40 0.9 0.8 0.5 38. ;;
- : float = 0.26
```

```
# frequenceCC 40 0.9 0.8 0.5 39. ;;
- : float = 0.27
# frequenceCC 40 0.9 0.8 0.5 40. ;;
- : float = 0.93

# frequenceCC 30 0.9 0.8 0.4 25. ;;
- : float = 1.
# frequenceCC 30 0.9 0.8 0.4 28. ;;
- : float = 1.
# frequenceCC 30 0.9 0.8 0.4 29. ;;
- : float = 0.28
# frequenceCC 30 0.9 0.8 0.4 30. ;;
- : float = 0.9
```

Lorsque B est proche de N, les fréquences sont faibles. Sinon, on a des fréquences égales à 1. On en déduit la propriété suivante :

Propriété 5.7

Pour $B \ll N$, les partitions optimales sont des combinaisons consécutives de deux entiers (au sens large), *i.e.* la taille des groupes ne diffère (au pire) que de 1.

Dans la suite, on évite de prendre B proche de N (à vrai dire, cette hypothèse est sans importance car l'objectif du group testing est justement de faire moins de tests que N).

On se repose alors la même question que lorsqu'on avait un test parfait : si les meilleures conceptions sont dans ce cas les combinaisons consécutives, ne seraient elles pas aussi les mêmes que celles pour des groupes de même taille ?

On écrit comme avant une fonction `frequenceMemeConc` de signature `int -> float -> float -> float -> float -> float` qui prend en paramètre N, Se, Sp, λ et B et qui pour 100 valeurs de p réparties uniformément dans $]0;1[$ renvoie la fréquence des cas où la conception idéale pour des groupes quelconques est la même que celle pour des groupes de même taille (à 1 près au pire).

Code source 5.7 : calcul de la fréquence des cas où les partitions optimales pour des groupes de même taille et des groupes de taille quelconque sont les mêmes, en OCaml

```
(* CombCons *)
let comb_cons n q r =
  let rec calcul n r =
    if n - r > q then
      calcul (n - 1) (r + q)
    else (n, r)
```

```
  in
  let (np, rp) = calcul n r in
    let alpha = np - 1
    and u = np - rp
    and v = q + rp - np + 1 in
  (alpha, u, v) ;;


(* Construction d'une liste de même élément *)
let rec listeMemesEl nb el =
  match nb with
  | 0 -> []
  | nb -> el :: (listeMemesEl (nb - 1) el) ;;


(* Calcul de la conception dans le cas même taille *)
let conceptionMemeTaille nPop p se sp b =
  let rec calcul_n n =
    match n with
    | n when n > nPop -> failwith "Pas de partition optimale pour
    ↪  les mêmes groupes"
    | n -> (
      let q = nPop / n and r = nPop mod n in
      let (alpha, u, v) = comb_cons n q r in
      let t1 = listeMemesEl u alpha
      and t2 = listeMemesEl v (alpha+1) in
      let part = t1 @ t2 in
      let et = espT part p se sp in
      if et <= b then part
      else calcul_n (n+1)
    )
  in
  calcul_n 2 ;;


(* Comparaison cas même taille / taille quelconque *)
let comparer nPop p se sp lambda b =
  try (
    let c_qcq = fst_tr (conceptionOpt nPop p se sp lambda b)
    and c_mt = conceptionMemeTaille nPop p se sp b in
    c_qcq = c_mt
```

```
  )
  with
    | Failure s -> failwith s ;;


let frequenceMemeConc nPop se sp lambda b =
  let rec comptage nb eff ps =
    match ps with
    | [] -> (nb, eff)
    | p :: t -> try (
        if comparer nPop p se sp lambda b then
          comptage (nb + 1) (eff + 1) t
        else comptage nb (eff + 1) t
      )
      with
        | _ -> comptage nb eff t
  in
  let f i =
    let i_f = Float.of_int i in (i_f +. 1.) /. 101.
  in
  let (nb, eff) = List.init 100 f |> (comptage 0 0) in
  try
    (Float.of_int nb) /. (Float.of_int eff)
  with
    | Division_by_zero -> failwith "Rien n'a été comparé" ;;
```

Présentons quelques résultats :

```
# frequenceMemeConc 50 0.9 0.8 0.5 50. ;;
- : float = 0.
# frequenceMemeConc 50 0.9 0.8 0.5 40. ;;
- : float = 0.0666666666666666657
# frequenceMemeConc 30 0.9 0.8 0.5 30. ;;
- : float = 0.
# frequenceMemeConc 30 0.9 0.8 0.5 20. ;;
- : float = 0.44444444444444442
```

Ces fréquences sont clairement faibles, voire très faibles pour certains cas : on ne peut pas confondre les deux conceptions.

En conclusion, comme avant, on va se contenter de parcourir les combinaisons consécutives pour optimiser la recherche de la meilleure conception.

5.3.3. Calcul plus efficace de la partition optimale

On a déjà un algorithme de calcul de l'ensemble des combinaisons consécutives (*cf.* sous-sous-section 4.2.4). L'algorithme de calcul se déduit facilement :

Code source 5.8 : détermination plus efficace de la conception optimale pour des groupes de taille quelconque, en OCaml

```
(* calcul de toutes les CC *)
let ensembleCC nPop =
  let rec calcul s =
    match s with
    | 0 -> []
    | s -> (
      let q = nPop / s and r = nPop mod s in
      let t1 = listeMemesEl (s-r) q
      and t2 = listeMemesEl r (q+1) in
      (t1 @ t2) :: calcul (s-1)
    )
  in
  calcul nPop ;;


(* calcul de la conception optimale pour les CC *)
let conceptionOptEff nPop p se sp lambda b =
  let listePart = ensembleCC nPop in
  let rec rechercheMin l partAct minAct trouve =
    match l with
    | [] when trouve -> (
        let e = espT partAct minAct se sp in
        (partAct, minAct, e)
      )
    | [] -> failwith "La partition optimale pour des groupes
    ↪  quelconques n'existe pas"
    | part :: t -> let et = espT part p se sp in
      if et > b then rechercheMin t partAct minAct trouve
      else (
        let efp = espFP part p se sp
        and efn = espFN part p se sp in
        let c = lambda *. efn +. (1. -. lambda) *. efp in
        if c < minAct then
          rechercheMin t part c true
        else rechercheMin t partAct minAct trouve
      )
```

```
    in
    rechercheMin listePart [] max_float false ;;
```

On vérifie qu'on a bien les mêmes résultats :

```
# conceptionOpt 60 0.1 0.9 0.8 0.5 45. ;;
- : int list * float * float =
([2; 2; 2; 2; 2; 2; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3],
 2.30015999999999821, 44.7015999999999849)
# conceptionOptEff 60 0.1 0.9 0.8 0.5 45. ;;
- : int list * float * float =
([2; 2; 2; 2; 2; 2; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3],
 2.30015999999999821, 44.7015999999999849)
# conceptionOpt 40 0.2 0.9 0.8 0.5 30. ;;
Exception:
Failure "La partition optimale pour des groupes quelconques n'existe pas".
# conceptionOptEff 40 0.2 0.9 0.8 0.5 30. ;;
Exception:
Failure "La partition optimale pour des groupes quelconques n'existe pas".
```

On calcule `conceptionOptEff 2000 0.1 0.9 0.8 0.5 1900.` en 0,57 s.

Cet algorithme plus efficace conclut cette partie. On va dans la suite s'intéresser aux populations hétérogènes.

Troisième partie
Population hétérogène

On considère maintenant la population hétérogène. On rappelle que l'on se donne les p_i, probabilités que l'individu i soit malade, avec $p_1 \leqslant p_2 \leqslant \cdots \leqslant p_N$.

On ne parle plus de partitions de l'entier N ; on considère maintenant les partitions de l'ensemble $[\![1; N]\!]$.

On ne va pas reprendre le même plan que précedemment. On va considérer le cas général et prendre Se, Sp quelconques, éventuellement tous les deux égaux à 1.

6. Parcours de l'ensemble des partitions

Comme on a maintenant N paramètres au lieu de 1 pour les probabilités d'être malade, on ne va pas s'intéresser à former des groupes de même taille car l'étude de ce cas est déjà trop complexe. On va directement considérer le cas général avec des groupes de taille quelconque, et on cherche à écrire un algorithme de calcul de la partition idéale.

L'idée de l'algorithme est de parcourir l'ensemble des partitions de $[\![1;N]\!]$ pour trouver la partition idéale selon un critère de sélection donné. Dans cette section, on va étudier l'élaboration de l'ensemble des partitions.

6.1. Création de l'ensemble des partitions

On veut créer un programme qui calcule l'ensemble des partitions de $[\![1;N]\!]$. On va écrire notre programme en OCaml.

On représente une partition Γ par une liste de listes d'entier. Chaque liste d'entier de la liste représente un groupe, dont les entiers sont les éléments de ce groupe. Par exemple, la partition

$$\Gamma = \{\{1;2;3\};\{4;6;7\};\{5;9\};\{8\}\}$$

est représentée par la liste suivante :

```
[[1; 2; 3]; [4; 6; 7]; [5; 9]; [8]]
```

On va créer l'ensemble des partitions de $[\![1;N]\!]$ récursivement.

Le cas de base est le suivant : l'ensemble des partitions de $[\![1;1]\!] = \{1\}$ est $\Big\{\{\{1\}\}\Big\}$, représenté par `[[[1]]]`.

Remarque. La première accolade (ou crochet) représente l'ensemble des partitions. La deuxième représente la seule partition considérée et la troisième le seul groupe de cette partition composé d'un seul élément qui est 1.

Supposons avoir calculé l'ensemble des partitions de $[\![1;N-1]\!]$. Pour calculer celui pour $[\![1;N]\!]$, on procède comme suit :

- on prend chaque partition $\Gamma_{N-1} = \{g_1,\dots,g_k\}$ de $[\![1;N-1]\!]$;
- on en crée les $k+1$ partitions de $[\![1;N]\!]$ suivantes : $\Gamma_{N-1} \cup \{\{N\}\}$, et les partitions $\{g_1,\dots,g_i \cup \{N\},\dots,g_k\}$ pour i entre 1 et k (autrement dit, on ajoute N à un groupe de Γ_{N-1}, et ce pour tous les groupes).

On obtient bien toutes les partitions de $[\![1;N]\!]$. En effet, on obtient bien une unique partition en rajoutant N comme ci-dessus (si deux partitions obtenues ci-dessus sont égales, alors les partitions dont on a retiré N sont aussi les mêmes). Et on les a toutes car il s'agit de toutes les manières dont on peut insérer l'élément N dans les partitions de $[\![1;N-1]\!]$ et car une partition de $[\![1;N]\!]$ est une partition de $[\![1;N-1]\!]$ dont on a rajouté l'élément N.

On en déduit un programme en OCaml déterminant l'ensemble des partitions de $[\![1;N]\!]$:

Code source 6.1 : calcul de l'ensemble des partitions de $[\![1;N]\!]$

```
#thread ;;
#require "core" ;;
```

```
(* le module Core sert à avoir des fonctions récursives terminales
↪  pour ainsi ne pas avoir l'erreur Stack_overflow (on voudra
↪  estimer le temps d'exécution) *)


(* pour une partition part, cette fonction renvoie la liste de
↪  toutes les partitions que l'on peut créer à partir de part en
↪  ajoutant seulement l'élément n *)
let ajouteTout_el n part =
  let rec ajoute_aux deb acc l =
    match l with
    | [] -> (Core.List.append part [[n]]) :: acc
    | g :: t ->
      let newP = Core.List.append deb ((Core.List.append g [n]) ::
      ↪  t) in
      ajoute_aux (Core.List.append deb [g]) (newP :: acc) t
  in
  ajoute_aux [] [] part ;;


(* liste de l'ensemble des partitions de [|1;n|] *)
let liste_partitions n =
  let rec construire k acc =
    match k with
    | k when k >= n -> acc
    | k -> let l = Core.List.map acc ~f:(ajouteTout_el (k+1))
                   |> Core.List.concat in
      construire (k+1) l
  in
  construire 1 [[[1]]] ;;


(* on donne une version sans le module Core, mais non récursive
↪  terminale *)
let ajouteTout_el n part =
  let rec ajoute_aux deb acc l =
    match l with
    | [] -> (part @ [[n]]) :: acc
    | g :: t ->
      let newP = deb @ ((g @ [n]) :: t) in
```

```
      ajoute_aux (deb @ [g]) (newP :: acc) t
  in
  ajoute_aux [] [] part ;;


let liste_partitions n =
  let rec construire k acc =
    match k with
    | k when k >= n -> acc
    | k -> let l = List.map (ajouteTout_el (k+1)) acc
                   |> List.concat in
      construire (k+1) l
  in
  construire 1 [[[1]]] ;;
```

6.2. Inefficacité de l'algorithme et nombre de Bell

Évaluons l'efficacité de notre algorithme précédent.

Définition 6.1

Le n^{e} nombre de Bell est le nombre de partitions de l'ensemble $[\![1;n]\!]$. On le note B_n.

On mesure le temps d'exécution de notre algorithme avec la même méthode que précédemment.

N	B_N	temps d'exécution
6	203	0,12 ms
7	877	0,46 ms
8	4 140	15 ms
9	21 147	52 ms
10	115 975	0,31 s
11	678 570	1,7 s
12	4 213 597	21 s
13	27 644 437	1 min 32 s
14	190 899 322	30 min 54 s

Le temps d'exécution n'est raisonnable que pour $N \leqslant 13$, ce qui est très faible comme taille de population. Cet algorithme est trop lent.

Quelque soit le critère de sélection, l'algorithme de détermination de la conception idéale par parcours de l'ensemble des partitions est très inefficace. On doit trouver un autre moyen.

7. Un algorithme plus efficace

Nous allons présenter un algorithme beaucoup efficace que le parcours naïf de l'ensemble des partitions. Il provient de l'article [4].

7.1. Critères de sélection

Cet algorithme permet d'englober les deux cas du test parfait et non parfait (on verra pourquoi plus tard). Dans le cas non parfait, le critère de sélection est le suivant : on cherche une partition Γ minimisant $\lambda\mathbb{E}(FN(\Gamma)) + (1-\lambda)\mathbb{E}(FP(\Gamma))$ et vérifiant les conditions $\mathbb{E}(T(\Gamma)) \leqslant B$ et $\forall g \in \Gamma$, $|g| \leqslant l$ (où $B \in \mathbb{R}^*$ et $l \in \mathbb{N}^*$). Dans le cas parfait, on cherche Γ minimisant $\mathbb{E}(T(\Gamma))$ et vérifiant aussi $\forall g \in \Gamma$, $|g| \leqslant l$.

En fait, ce sont les deux critères qu'on a vu précedemment mais dont on a rajouté l'hypothèse de ne pas avoir des groupes trop grands (voir la sous-section 5.2 pour la raison pour laquelle on n'a pas considéré cette hypothèse dans le cas homogène). On peut voir les critères de majoration par B et l comme des limitations des ressources disponibles (nombre de tests, taille maximale du mélange des échantillons...). On peut aussi voir l comme une limite au delà de laquelle la procédure ne marche pas (si on teste trop de personnes d'un coup, on risque de ne voir une personne malade parmi beaucoup de personnes saines).

Il faut savoir comment calculer les espérances ci-dessus. Pour cela, on a des formules comme dans le cas homogène.

7.2. Formules des espérances du nombre de test, de faux positifs et de faux négatifs

On admet ces formules dont on trouve la démonstration dans l'article [3].

7.2.1. Nombre de test

Propriété 7.1 : formule de l'espérance du nombre de test dans le cas hétérogène

$$\mathbb{E}(T(\Gamma)) = |\Gamma| + \sum_{\substack{g\in\Gamma \\ |g|\neq 1}} |g| \left(Se - (Se + Sp - 1) \prod_{i\in g} (1-p_i) \right).$$

7.2.2. Nombre de faux positifs

Propriété 7.2 : formule de l'espérance du nombre de faux positifs dans le cas hétérogène

$$\mathbb{E}(FP(\Gamma)) = (1-Se)\sum_{\substack{g\in\Gamma\\ g=\{i\}}}(1-p_i) + (1-Sp)\sum_{\substack{g\in\Gamma\\ |g|\neq 1}}\left(Se\sum_{i\in g}(1-p_i) - |g|(Se+Sp-1)\prod_{i\in g}(1-p_i)\right).$$

7.2.3. Nombre de faux négatifs

Propriété 7.3 : formule de l'espérance du nombre de faux négatifs dans le cas hétérogène

$$\mathbb{E}(FN(\Gamma)) = (1-Se)\sum_{\substack{g\in\Gamma\\ g=\{i\}}} p_i + (1-Se^2)\sum_{\substack{g\in\Gamma\\ |g|\neq 1}}\sum_{i\in g} p_i.$$

7.3. Algorithme

7.3.1. Principe

Définition 7.1

Une partition $\Gamma = \{g_1, \ldots, g_k\}$ de $[\![1; N]\!]$ est ordonnée si et seulement si :

$$\exists \sigma \in \mathfrak{S}_k,\ \forall i, j \in [\![1; k]\!],\ i < j \Rightarrow \big(\forall x \in g_{\sigma(i)},\ \forall y \in g_{\sigma(j)},\ x < y\big).$$

Une autre définition équivalente, et plus claire, est la suivante : Γ est ordonnée si et seulement si il existe $p \in \mathbb{N}$ et $\alpha_1 < \alpha_2 < \cdots < \alpha_p \in [\![1; N]\!]$ tels que :

$$\Gamma = \{[\![1; \alpha_1]\!], [\![\alpha_1 + 1; \alpha_2]\!], \ldots, [\![\alpha_{p-1} + 1; \alpha_p]\!], [\![\alpha_p + 1; N]\!]\}.$$

Exemple 7.1

La partition $\{\{1;2;3\};\{4;6;7\};\{5;9\};\{8\}\}$ n'est pas ordonnée.
La partition $\{\{1;2;3\};\{4;5;6;7\};\{8\};\{9\}\}$ est ordonnée.
Et $\{\{1;2;3\};\{4;5;6;7\};\{9\};\{8\}\}$ est aussi ordonnée.

Propriété 7.4

Il y a exactement 2^{N-1} partitions ordonnées de $[\![1;N]\!]$.

Démonstration. Pour créer une partition ordonnée de $[\![1;N]\!]$, on écrit $1, 2, \dots, N$ puis on place des barres entre les nombres. On en déduit les groupes de la partition : on en forme un avec les nombres qui ne sont pas séparés par une barre. Par exemple, $1|2,3,4|5,6$ forme la partition ordonnée $\{\{1\},\{2,3,4\},\{5,6\}\}$.
Or, il y a 2^{N-1} façons de placer ces barres (cela revient à choisir un $(N-1)$-uplet dont les éléments sont 0 (pas de barre) ou 1 (avec barre)). □

Remarque. Ce nombre correspond aussi au nombre de compositions de N (partitions de N où l'on impose aucun ordre). En particulier, il y a plus de partitions ordonnées de $[\![1;N]\!]$ qu'il n'y a de partitions de N. Et il y a moins de partitions ordonnées qu'il n'y a de partitions de $[\![1;N]\!]$.

Remarque. On peut décrire une partition ordonnée juste avec la taille des groupes, comme dans le cas homogène. Mais cette fois-ci, on n'a pas besoin d'imposer un ordre décroissant pour les éléments. Par exemple, $(1,3,2)$ représente $\{\{1\},\{2,3,4\},\{5,6\}\}$.

Voici une propriété sur les partitions optimales, fondamentale pour l'algorithme qui suit.

Propriété 7.5

En cas d'existence d'une partition optimale, il existe une partition optimale ordonnée.

On admet cette propriété (*cf.* démonstration dans l'article [4]).

Cette propriété est très importante : on a plus besoin de parcourir l'ensemble des partitions de $[\![1;N]\!]$ mais seulement les partitions ordonnées.

Définition 7.2

Pour un groupe donnée g d'une partition Γ, on notera $T_{\text{gr}}(g)$, $FP_{\text{gr}}(g)$ et $FN_{\text{gr}}(g)$ les variables aléatoires associants respectivement le nombre de tests, de faux positifs et de faux négatifs pour le groupe g à l'issue de la procédure.

Remarque. Ainsi, on a :

$$T(\Gamma) = \sum_{g\in\Gamma} T_{\text{gr}}(g), \qquad FP(\Gamma) = \sum_{g\in\Gamma} FP_{\text{gr}}(g), \qquad \text{et} \qquad FN(\Gamma) = \sum_{g\in\Gamma} FN_{\text{gr}}(g).$$

En fait, c'est avec ces variables aléatoires qu'on démontre les formules des espérances.

Introduisons le graphe orienté $G = (V, A)$ suivant :
— l'ensemble des sommets est $V = [\![1; N+1]\!]$;
— l'ensemble des arcs est :

$$A = \Big\{(i,j) \,\big|\, 1 \leqslant i < j \leqslant N+1;\ \mathbb{E}(T_{\text{gr}}([\![i, j-1]\!])) \leqslant B;\ j - i \leqslant l\Big\}.$$

L'individu $N+1$ est dit artificiel. On lui associe une probabilité d'être malade de 1. On rend le graphe pondéré comme suit : pour $(i,j) \in A$, le poids de l'arc est :

$$C_{ij} = \lambda\,\mathbb{E}(FN_{\text{gr}}([\![i, j-1]\!])) + (1-\lambda)\mathbb{E}(FP_{\text{gr}}([\![i, j-1]\!])).$$

On associe aussi à cette arc un nombre appelé contrainte :

$$R_{ij} = \mathbb{E}(T_{\text{gr}}([\![i, j-1]\!]))$$

Un chemin entre les sommets 1 et $N+1$ correspond à une partition ordonnée. Si $1 \to s_1 \to \cdots \to s_k \to N+1$ est un chemin (avec des sommets deux à deux distincts et croissants), alors celui-ci correspond à la partition ordonnée suivante :

$$\Gamma = \{[\![1; s_1 - 1]\!], [\![s_1; s_2 - 1]\!], \ldots, [\![s_{k-1}; s_k - 1]\!], [\![s_k; N]\!]\}.$$

Ainsi, en posant $s_0 = 1$ et $s_{k+1} = N+1$, le coût total C du chemin est en fait (par linéarité de l'espérance) :

$$C = \sum_{i=0}^{k} C_{s_i s_{i+1}} = \lambda\,\mathbb{E}(FN(\Gamma)) + (1-\lambda)\mathbb{E}(FP(\Gamma)).$$

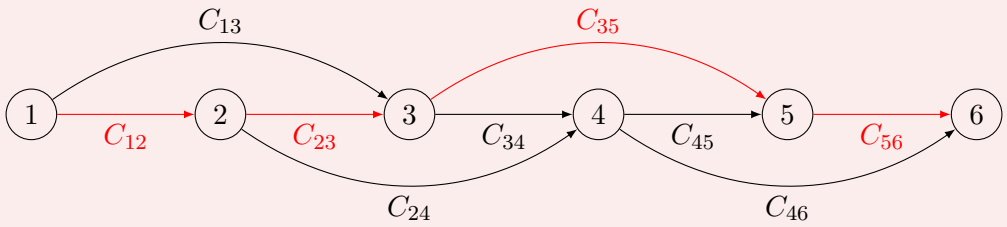
On notera aussi que :

$$R = \sum_{i=0}^{k} R_{s_i s_{i+1}} = \mathbb{E}(T(\Gamma)).$$

Notre graphe G représente alors l'ensemble des partitions ordonnées que l'on doit considérer. En fait, la détermination de la partition optimale est un problème du plus court chemin, mais avec contraintes.

Exemple 7.2

Donnons un exemple graphique. Prenons $N = 5$, $l = 2$ et évitons de considérer la condition sur B.
Le chemin en rouge représente la partition $\{\{1\}, \{2\}, \{3,4\}, \{5\}\}$, dont le coût vaut $C = C_{12}+C_{23}+C_{35}+C_{56}$. La consommation des tests est $R = R_{12}+R_{23}+R_{35}+R_{56}$.



Définition 7.3

Pour un chemin entre les sommets 1 et $i \in S$, on définit récursivement la notion d'étiquette.

- Si $i = 1$, alors l'étiquette est un quadruplet $(\text{NIL}, \text{NIL}, 0, 0)$;
- Sinon, l'étiquette est $\ell = (\ell_{\text{pred}}, i_{\text{pred}}, C, R)$ où :
  - ℓ_{pred} est l'étiquette du même chemin mais dont on a enlevé le dernier arc (i_{pred}, i) ;
  - C est le coût du chemin : $C(\ell) = C(\ell_{\text{pred}}) + C_{i_{\text{pred}}i}$;
  - R est la consommation des ressources (nombre de tests utilisés) : $R(\ell) = R(\ell_{\text{pred}}) + R_{i_{\text{pred}}i}$.

Exemple 7.3

En reprenant le graphe précédent, l'étiquette du chemin formé par les 3 premiers arcs rouges est :

$$\ell = (\ell_2, 3, C_{12} + C_{23} + C_{35}, R_{12} + R_{23} + R_{35})$$

avec $\ell_2 = (\ell_1, 2, C_{12} + C_{23}, R_{12} + R_{23})$ et $\ell_1 = ((\text{NIL}, \text{NIL}, 0, 0), 1, C_{12}, R_{12})$.

Définition 7.4

Pour deux étiquettes ℓ_1 et ℓ_2 dont les chemins finissent par le même sommet, on note $\ell_1 \prec \ell_2$ lorsque $C(\ell_1) \leqslant C(\ell_2)$. C'est bien sûr une relation d'ordre.

Définition 7.5

Soient deux étiquettes ℓ_1 et ℓ_2. On dit que ℓ_1 domine ℓ_2 lorsque $\ell_1 \prec \ell_2$ et $R(\ell_1) \leqslant R(\ell_2)$.

Remarque. Cette dernière définition permet d'améliorer l'efficacité de l'algorithme. C'est surtout grâce à cette relation qu'on peut fusionner les deux critères selon la nature du test.

Définition 7.6

Soit $j \in S$. On définit récursivement un ensemble Λ_j d'étiquettes dont leur chemin finit par j, ordonné selon $\prec$, par :

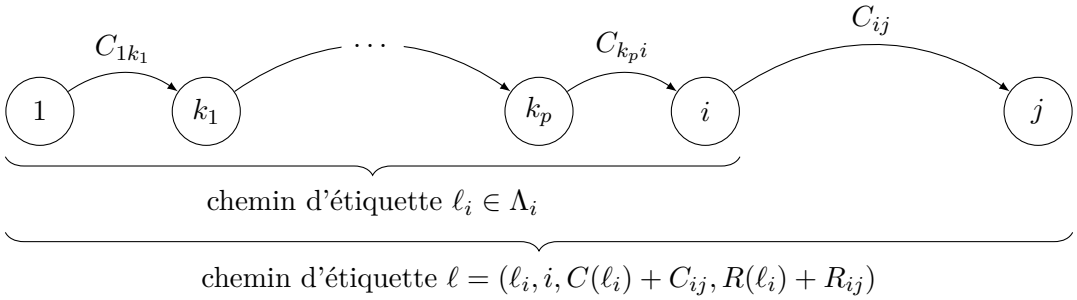
- si $j = 1$, alors $\Lambda_1 = \{(\text{NIL}, \text{NIL}, 0, 0)\}$;
- sinon, on initialise Λ_j à l'ensemble vide, puis pour chaque $i \in [\![1; j-1]\!]$ tel que $(i, j) \in A$:
  - pour chaque ℓ_i dans Λ_i, on calcule l'étiquette ℓ du chemin formé par

celui de ℓ_i et par l'arc (i,j) :

$$\ell = (\ell_i, i, C(\ell_i) + C_{ij}, R(\ell_i) + R_{ij});$$

— si $R(\ell_i) + R_{ij} \leqslant B$, et si ℓ n'est pas dominée par une étiquette de Λ_j, on l'ajoute dans Λ_j (toujours selon l'ordre $\prec$) puis on enlève de cet ensemble les étiquettes qui sont dominées par ℓ.
— si $R(\ell_i) + R_{ij} > B$ ou si ℓ est dominée par une étiquette de Λ_j, on ne l'ajoute pas.

Illustrons la construction récursive de Λ_j par un schéma :



Notons $\Lambda_j = \left\{{\ell_1}^j, \ldots, {\ell_m}^j\right\}$ et supposons $R(\ell_i) + R_{ij} \leqslant B$ (dans le cas contraire, on ne fait rien).

— Si ℓ est dominée par un ${\ell_a}^j$, *i.e.* si ${\ell_a}^j \prec \ell$ et $R\left({\ell_a}^j\right) \leqslant R(\ell_i) + R_{ij} = R(\ell)$, on n'ajoute pas ℓ ;
— Si ℓ n'est dominée par aucun des ${\ell_a}^j$, *i.e.*

$$\forall {\ell_a}^j \in \Lambda_j, \quad {\ell_a}^j \not\prec \ell \qquad \text{ou} \qquad R\left({\ell_a}^j\right) > R(\ell_i) + R_{ij} = R(\ell),$$

alors on insère ℓ :

$$\Lambda_j = \{\underbrace{{\ell_1}^j, \ldots, {\ell_q}^j}_{{\ell_a}^j \prec \ell}, \ell, \underbrace{{\ell_{q+1}}^j, \ldots, {\ell_m}^j}_{\ell \prec {\ell_a}^j}\}.$$

Or, ${\ell_a}^j$ est dominée par $\ell \iff \ell \prec {\ell_a}^j$ et $R(\ell) \leqslant R\left({\ell_a}^j\right)$. On enlève donc les étiquettes après ℓ telles que $R(\ell) \leqslant R\left({\ell_a}^j\right)$ (celles avant ne peuvent être dominées).

Remarque. Fondamentalement, on purrait se contenter de seulement rajouter les étiquettes telles que $R(\ell_i) + R_{ij} \leqslant B$. En fait, cette règle de dominance sert à deux choses :
— réduire le nombre d'étiquettes à traiter (on ne veut pas garder les étiquettes qui donnent à la fois un coût C et une consommation des ressources R pires que l'étiquette qu'on rajoute) ;
— prendre en compte le critère de sélection dans le cas des tests parfaits : dans ce cas, les coûts C sont tous nuls. On ne prend que les étiquettes qui donnent un R minimal : c'est le critère que l'on veut.

Propriété 7.6

Le chemin optimal donnant la partition optimale ordonnée est donné par la première étiquette de Λ_{N+1}. Si Λ_{N+1} est vide, c'est que la partition optimale n'existe pas.

Démonstration. L'ensemble Λ_{N+1} est composé d'étiquettes de chemins commençant par 1 et terminant par $N+1$. Elles représentent donc bien une partition de $[\![1;N]\!]$ ordonnée.
Or, pour tout j dans $[\![1;N+1]\!]$, Λ_j est l'ensemble des étiquettes des chemins entre 1 et j tels que $R \leqslant B$. En effet, pour ajouter une étiquette dans Λ_j (si $j > 1$), il faut que R soit inférieur ou égal à B.
De plus, chaque arc est dans A donc les groupes sont de taille inférieure ou égale à l.
Enfin, Λ_j est construit de tel sorte à trier les étiquettes selon la relation $\prec$. Or, on a vu que pour Λ_{N+1}, $C = \lambda\mathbb{E}(FN(\Gamma)) + (1-\lambda)\mathbb{E}(FP(\Gamma))$. Donc la première étiquette correspond à la partition optimale. □

L'idée de l'algorithme est donc de construire progressivement les ensembles Λ_j jusqu'à Λ_{N+1} pour en déduire la partition voulue.

7.3.2. Codage

Comme les vecteurs dans OCaml sont indexés à partir de 0, on a préféré raisonner avec des indices allant de 0 à N au lieu de 1 à $N+1$. Cependant, l'algorithme qui suit revoie bien une partition dont les éléments sont dans $[\![1;N]\!]$.

Remarque. Les probabilités d'être malade pour chaque individu sont stockées dans un tableau noté `r`. Cette notation, comme la majorité des autres, provient des sources bibliographiques.

Une étiquette sera représentée par le type suivant :

```
type label =
  | Nil
  | L of {pred : int * label ; c : float ; r : float}
```

On va écrire deux fonctions auxiliaires : l'une prend la liste des prédécesseurs d'une étiquette d'un chemin de 1 à $N+1$ et renvoie la partition ordonnée optimale. L'autre prend une liste d'étiquettes Λ_j et une étiquette ℓ, et renvoie la liste Λ_j dont on a inséré (ou non) ℓ selon le processus expliqué ci-dessus.

On écrit aussi d'autres fonctions auxiliaires permettant le calcul des espérances et la comparaison d'étiquettes.

Voici l'algorithme :

Code source 7.1 : Calcul plus efficace de la partition optimale

```
type partition = int list list ;;


type label =
  | Nil
  | L of {pred : int * label ; c : float ; r : float}


(*
  subjects : int, représente [1;...;N]
  init_el : liste des entiers moins 1 qui commencent dans chaque
↪  partition
  par exemple, recover_partition 10 [0;3;7] renvoie
↪  [[1;2;3],[4;5;6;7],[8;9;10]]
*)
let recover_partition subjects init_el =
  let rec cummul initial final =
    match initial with
    | k when k = final -> [final]
    | k -> k :: (cummul (k + 1) final)
  in
  let rec construire list_init =
    match list_init with
    | [] -> []
    | [x] -> [cummul (x+1) subjects]
    | deb :: fin :: t ->
      (cummul (deb+1) fin) :: (construire (fin :: t))
  in
  construire init_el ;;


(* deux fonctions auxilliares de comparaison *)
let compare_c (l1 : label) (l2 : label) =
  match (l1, l2) with
  | (Nil, _) -> failwith "Nil"
  | (_, Nil) -> failwith "Nil"
  | (L x1,L x2) -> x1.c <= x2.c ;;


let dominates (l1 : label) (l2 : label) =
  match (l1, l2) with
```

```
  | (Nil, _) -> failwith "Nil"
  | (_, Nil) -> failwith "Nil"
  | (L x1,L x2) ->
    assert (x1.c <= x2.c) ;
    x1.r <= x2.r ;;


let extend_and_dominance (lambda_j : label list) (new_l : label) =
  let rec inserer lamb index =
    match lamb with
    | [] -> ([new_l], index)
    | x :: t as l when compare_c new_l x -> (new_l :: l, index)
    | x :: t -> if dominates x new_l then raise Exit
      else let (l, i) = inserer t (index + 1) in (x :: l, i)
  in
  let rec retirer lamb i index =
    match lamb with
    | [] -> []
    | x :: t when i <= index -> x :: (retirer t (i+1) index)
    | x :: t -> if dominates new_l x then retirer t i index else x
    ↪  :: (retirer t i index)
  in
  try
    let (lamb, index) = inserer lambda_j 0 in
    retirer lamb 0 index
  with
  | Exit -> lambda_j ;;


(* on considère le groupe [i;i+1;...;j-1] dont on va calculer
↪  l'espérance de faux négatifs *)
let esperance_fn i j (r : float array) se =
  let n = j - i in
  let rec somme_el_id deb fin =
    match deb with
    | deb when deb = fin -> r.(deb)
    | deb -> r.(deb) +. (somme_el_id (deb + 1) fin)
  in
  match n with
  | 1 -> (1. -. se) *. r.(i)
  | n -> let s = somme_el_id i (j-1) in (1. -. se *. se) *. s ;;
```

```
(* fonctions auxilliares de somme et produit pour l'espérance de
↪  faux positifs *)
let rec somme_el v deb fin =
    match deb with
    | deb when deb = fin -> 1. -. v.(deb)
    | deb -> (1. -. v.(deb)) +. (somme_el v (deb + 1) fin) ;;


let rec prod_el v deb fin =
    match deb with
    | deb when deb = fin -> 1. -. v.(deb)
    | deb -> (1. -. v.(deb)) *. (prod_el v (deb + 1) fin) ;;


let esperance_fp i j (r : float array) se sp =
  match j - i with
  | 1 -> (1. -. sp) *. (1. -. r.(i))
  | n -> let s = somme_el r i (j-1) and p = prod_el r i (j-1) in
    (1. -. sp) *. se *. s -. (Float.of_int n) *. (1. -. sp) *. (se
    ↪  +. sp -. 1.) *. p ;;


let esperance_t i j (r : float array) se sp =
  match j - i with
  | 1 -> 1.
  | n -> let p = prod_el r i (j-1) in 1. +. (Float.of_int n) *. (se
  ↪  -. (se +. sp -. 1.) *. p) ;;


(* crée l'ensemble A *)
let arc_set (subjects : int) (b : float) (l : int) (r : float
↪  array) se sp =
  let node_set = subjects + 1 in
  let rec cons_j i j =
    match j with
    | j when j = node_set -> []
    | j when (j - i <= l) && (esperance_t i j r se sp <= b) ->
    ↪  (i,j) :: (cons_j i (j+1))
    | j -> cons_j i (j+1)
```

```
  in
  let rec cons_i i =
    match i with
    | i when i = node_set -> []
    | i -> (cons_j i (i+1)) @ (cons_i (i+1))
  in cons_i 0 ;;


(* renvoie un tableau auquel on a rajouté 1. à la fin *)
let add_artificial r =
  let aux_init n i =
    if i = n then 1. else r.(i)
  in
  let n = Array.length r in
  Array.init (n + 1) (aux_init n) ;;


let gtcspp (r : float array) se sp (small_lambda : float) (b :
↪  float) (l : int) =
  let subjects = Array.length r in
  let r_exp = add_artificial r in
  let arcs = arc_set subjects b l r_exp se sp in
  let lambdas = Array.make (subjects + 1) [] in
  lambdas.(0) <- [Nil] ;
  for j = 1 to subjects do
    for i = 0 to j - 1 do
      if List.mem (i,j) arcs then begin
        let r_ij = esperance_t i j r_exp se sp in
        let e_fn = esperance_fn i j r_exp se and e_fp =
        ↪  esperance_fp i j r_exp se sp in
        let c_ij = small_lambda *. e_fn +. (1. -. small_lambda) *.
        ↪  e_fp in
        let aux_iter el =
          match el with
          | Nil -> if r_ij <= b then (
              let l_prime = L {pred = (i,Nil) ; c = c_ij ; r =
              ↪  r_ij} in
              lambdas.(j) <- extend_and_dominance lambdas.(j)
              ↪  l_prime
            )
          | L x -> if x.r +. r_ij <= b then (
```

```
            let l_prime = L {pred = (i,el) ; c = x.c +. c_ij ; r
            ↪  = x.r +. r_ij} in
            lambdas.(j) <- extend_and_dominance lambdas.(j)
            ↪  l_prime
          )
      in
      List.iter aux_iter lambdas.(i)
    end ;
  done ;
done ;
try
  let first_els = ref [] in
  let lab = List.hd lambdas.(subjects) in
  let rec aux_construct lab =
    match lab with
    | Nil -> ()
    | L x -> (
        let (pre, lab_bis) = x.pred in
        first_els := pre :: (!first_els) ;
        aux_construct lab_bis
      )
  in
  aux_construct lab ;
  match lab with
  | Nil -> (* ce cas n'a pas trop de sens, il est seulement ici
              pour rendre le filtrage complet *)
    let res = recover_partition subjects !first_els in
    (res, 0., 0.)
  | L x ->
    let c_final = x.c and r_final = x.r in
    let res = recover_partition subjects !first_els in
    (res, c_final, r_final)
with
| Failure _ -> (* l'exception ne peut que provenir de List.hd *)
  failwith "La partition optimale n'existe pas" ;;
```

Remarque. L'acronyme `gtcspp` signifie « Group Testing Constrained Shortest Path Problem ».

Donnons quelques résultats donnés par l'algorithme :

```
# let r = Array.init 100 (fun x -> Random.float 0.3) in
gtcspp r 0.9 0.8 0.5 85. 10 ;;
- : int list list * float * float =
```

```
([[1; 2]; [3; 4]; [5; 6]; [7; 8]; [9; 10]; [11; 12]; [13; 14]; [15; 16; 17];
  [18; 19]; [20; 21]; [22; 23]; [24; 25]; [26; 27; 28]; [29; 30]; [31; 32];
  [33; 34; 35]; [36; 37]; [38; 39]; [40; 41]; [42; 43]; [44; 45; 46];
  [47; 48]; [49; 50; 51]; [52; 53]; [54; 55]; [56; 57; 58]; [59; 60; 61];
  [62; 63; 64]; [65; 66]; [67; 68]; [69; 70]; [71; 72; 73]; [74; 75];
  [76; 77]; [78; 79]; [80; 81]; [82; 83]; [84; 85; 86]; [87; 88]; [89; 90];
  [91; 92]; [93; 94]; [95; 96; 97]; [98; 99; 100]],
 4.1516066273336163, 84.779744153505419)

# let r = Array.init 100 (fun x -> Random.float 0.3) in
gtcspp r 0.9 0.8 0.5 75. 10 ;;
Exception: Failure "La partition optimale n'existe pas".

# let r = Array.init 100 (fun x -> Random.float 1.) in
gtcspp r 0.9 0.8 0.5 100. 10 ;;
- : int list list * float * float =
([[1]; [2]; [3; 4]; [5]; [6]; [7]; [8]; [9]; [10]; [11]; [12]; [13];
  [14; 15]; [16]; [17]; [18; 19]; [20; 21; 22]; [23]; [24]; [25]; [26];
  [27]; [28; 29]; [30]; [31]; [32]; [33]; [34]; [35]; [36; 37; 38]; [39; 40];
  [41]; [42]; [43]; [44; 45]; [46; 47]; [48]; [49]; [50; 51; 52]; [53];
  [54]; [55]; [56; 57]; [58]; [59]; [60]; [61]; [62]; [63]; [64; 65];
  [66; 67]; [68]; [69]; [70]; [71]; [72]; [73; 74]; [75]; [76]; [77];
  [78; 79]; [80; 81]; [82]; [83]; [84]; [85; 86]; [87; 88]; [89]; [90];
  [91; 92]; [93]; [94]; [95]; [96; 97]; [98]; [99]; [100]],
 6.05739710472439707, 99.6477314526820237)

# let r = Array.init 60 (fun x -> 0.1) in
gtcspp r 0.9 0.8 0.5 45. 10 ;;
- : int list list * float * float =
([[1; 2; 3]; [4; 5; 6]; [7; 8; 9]; [10; 11; 12]; [13; 14; 15]; [16; 17; 18];
  [19; 20; 21]; [22; 23; 24]; [25; 26; 27]; [28; 29; 30]; [31; 32; 33];
  [34; 35; 36]; [37; 38; 39]; [40; 41; 42]; [43; 44]; [45; 46]; [47; 48];
  [49; 50]; [51; 52]; [53; 54; 55]; [56; 57; 58]; [59; 60]],
 2.30015999999999821, 44.7015999999999849)

# let r = Array.init 100 (fun x -> Random.float 1.) in
gtcspp r 1. 1. 0.5 100. 10 ;;
- : int list list * float * float =
([[1]; [2]; [3; 4]; [5]; [6]; [7; 8; 9]; [10]; [11]; [12]; [13]; [14];
  [15]; [16]; [17]; [18]; [19]; [20; 21; 22]; [23]; [24]; [25]; [26];
  [27]; [28]; [29; 30]; [31]; [32]; [33]; [34]; [35]; [36]; [37]; [38; 39];
  [40]; [41]; [42]; [43]; [44]; [45]; [46; 47; 48]; [49]; [50; 51]; [52];
  [53]; [54]; [55]; [56]; [57]; [58]; [59; 60; 61]; [62]; [63]; [64; 65];
  [66]; [67]; [68]; [69; 70]; [71]; [72]; [73]; [74]; [75]; [76];
```

```
  [77; 78; 79]; [80]; [81]; [82]; [83]; [84]; [85]; [86]; [87]; [88];
  [89]; [90]; [91]; [92]; [93]; [94]; [95]; [96]; [97]; [98]; [99]; [100]],
 0., 95.0533207280443833)
```

Remarque. L'instruction `Random.float f` où `f` est un flottant renvoie un flottant aléatoire dans l'intervalle $[0; f]$.

On remarquera la cohérence des résultats avec ce qui a été fait précédemment lorsqu'on fixe une population homogène (*cf.* sous-sous-section 5.3.3 ; on a bien 6 groupes de 2 et le reste des groupes de 3, et les espérances sont les mêmes). On remarquera aussi que l'algorithme fonctionne pour des tests parfaits.

Avec d'autres exemples, on peut dégager une tendance générale sur la nature des groupes :

— les individus de grande probabilité d'être malade ont tendance à être dans des groupes individuels ;

— les populations dont les probabilités sont faibles ont tendance à former des groupes de grande taille (lorsque l est assez grand).

7.3.3. Vérification expérimentale de la correction de l'algorithme

Voici un algorithme qui à une partition Γ calcule une estimation du coût et de la contrainte associée à la procédure. Il s'agit d'utiliser la méthode de Monte-Carlo.

On simule l'état d'une population par une liste de listes d'entiers de même structure que la partition ordonnée, mais dont les éléments sont des 0 ou des 1 selon que l'invididu représenté par un élément d'une liste de la liste soit sain ou malade. Par exemple, si $\Gamma = \{\{1; 3\}; \{2; 4; 5\}\}$ est une partition représentée par `[[1;3];[2;4;5]]`, une simulation possible de l'état de la population est `[[0;0];[0;1;0]]`, signifiant que les individus $1, 2, 3, 5$ sont sains et que l'individu 4 est malade.

Code source 7.2 : calcul des valeurs approchées du coût et de la contrainte d'une partition

```
let rec presenceUn l =
  match l with
  | [] -> false
  | a :: t when a = 1 -> true
  | _ :: t -> presenceUn t ;;


let uneSimulation r partOpt =
  let rec constGroupe g =
    match g with
    | [] -> []
    | i :: t -> (
        let p = Random.float 1. in
```

```
        if p <= r.(i-1) then
          1 :: (constGroupe t)
        else
          0 :: (constGroupe t)
      )
  in
  let rec constSim l =
    match l with
    | [] -> []
    | g :: t -> (
        let simG = constGroupe g in
        simG :: (constSim t)
      )
  in
  constSim partOpt ;;


let rec nbTests sim se sp =
  match sim with
  | [] -> 0
  | simG :: t when presenceUn simG -> (
      let p = Random.float 1. in
      if p <= se then
        let n = List.length simG in
        1 + n + nbTests t se sp
      else
        1 + nbTests t se sp
    )
  | simG :: t -> (
      let p = Random.float 1. in
      if p <= 1. -. sp then
        let n = List.length simG in
        1 + n + nbTests t se sp
      else
        1 + nbTests t se sp
    ) ;;


let simulationTests r partOpt se sp nbSim =
  let rec sommeNbTests k =
    match k with
```

```
    | 0 -> 0
    | k -> (
        let sim = uneSimulation r partOpt in
        let n = nbTests sim se sp in
        n + (sommeNbTests (k-1))
      )
  in
  let somme = sommeNbTests nbSim |> Float.of_int in
  somme /. (Float.of_int nbSim) ;;


let rec compteUns l =
  match l with
  | [] -> 0
  | a :: t when a = 1 -> 1 + compteUns t
  | _ :: t -> compteUns t ;;


let rec coupleGroupeFNP simG se sp =
  match simG with
  | [] -> (0, 0)
  | 0 :: t -> (
      let (fn, fp) = coupleGroupeFNP t se sp in
      let p = Random.float 1. in
      if p <= sp then (fn, fp) else (fn, fp + 1)
    )
  | 1 :: t -> (
      let (fn, fp) = coupleGroupeFNP t se sp in
      let p = Random.float 1. in
      if p <= se then (fn, fp) else (fn + 1, fp)
    )
  | _ -> failwith "J'aime les filtrages complets" ;;


let rec coupleFNP sim se sp =
  match sim with
  | [] -> (0, 0)
  | simG :: t when presenceUn simG -> (
      let (fn, fp) = coupleFNP t se sp in
      let p = Random.float 1. in
      if p <= se then
```

```
        let (fnG, fpG) = coupleGroupeFNP simG se sp in
        (fn + fnG, fp + fpG)
      else
        let nbUns = compteUns simG in
        (fn + nbUns, fp)
    )
  | simG :: t -> (
      let (fn, fp) = coupleFNP t se sp in
      let p = Random.float 1. in
      if p <= 1. -. sp then
        let (fnG, fpG) = coupleGroupeFNP simG se sp in
        (fn + fnG, fp + fpG)
      else
        (fn, fp)
    ) ;;

let simulationCout r partOpt se sp lambda nbSim =
  let rec calculSomme k =
    match k with
    | 0 -> 0.
    | k -> (
        let sim = uneSimulation r partOpt in
        let (fn, fp) = coupleFNP sim se sp in
        let fn_F = Float.of_int fn
        and fp_F = Float.of_int fp in
        let c = lambda *. fn_F +. (1. -. lambda ) *. fp_F in
        c +. calculSomme (k-1)
      )
  in
  let nbSim_F = Float.of_int nbSim in
  (calculSomme nbSim) /. nbSim_F ;;


let simulation r partOpt se sp lambda nbSim =
  let sR = simulationTests r partOpt se sp nbSim in
  let sC = simulationCout r partOpt se sp lambda nbSim in
  (sC, sR) ;;
```

Cet algorithme est général : on peut prendre n'importe quelle partition Γ pour obtenir une estimation du coût et de la contrainte pour cette partition, mais ce qui nous intéresse ici est vérifier expérimentalement la correction de l'algorithme qui donne une partition donnée (celle optimale). De plus, on peut même prendre τ pour des populations

homogènes, sous réverse de convertir τ en Γ (τ donne les tailles des groupes de Γ).

Vérifions la correction de l'algorithme avec le quatrième exemple ci-dessus.

```
# let r = Array.init 60 (fun x -> 0.1) ;;
val r : float array =
  [|0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1;
    0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1;
    0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1;
    0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1; 0.1;
    0.1; 0.1; 0.1; 0.1|]
# let p =
[[1; 2; 3]; [4; 5; 6]; [7; 8; 9]; [10; 11; 12]; [13; 14; 15]; [16; 17; 18];
[19; 20; 21]; [22; 23; 24]; [25; 26; 27]; [28; 29; 30]; [31; 32; 33];
[34; 35; 36]; [37; 38; 39]; [40; 41; 42]; [43; 44]; [45; 46]; [47; 48];
[49; 50]; [51; 52]; [53; 54; 55]; [56; 57; 58]; [59; 60]] ;;
val p : int list list =
  [[1; 2; 3]; [4; 5; 6]; [7; 8; 9]; [10; 11; 12]; [13; 14; 15]; [16; 17; 18];
   [19; 20; 21]; [22; 23; 24]; [25; 26; 27]; [28; 29; 30]; [31; 32; 33];
   [34; 35; 36]; [37; 38; 39]; [40; 41; 42]; [43; 44]; [45; 46]; [47; 48];
   [49; 50]; [51; 52]; [53; 54; 55]; [56; 57; 58]; [59; 60]]
# simulation r p 0.9 0.8 0.5 1000 ;;
- : float * float = (2.285, 44.557)
# simulation r p 0.9 0.8 0.5 1000 ;;
- : float * float = (2.315, 44.577)
# simulation r p 0.9 0.8 0.5 1000 ;;
- : float * float = (2.282, 44.841)
# simulation r p 0.9 0.8 0.5 10000 ;;
- : float * float = (2.29155, 44.7555)
```

Les estimations sont cohérentes.

Regardons un autre exemple.

```
# let r = Array.init 200 (fun x -> Float.of_int x /. 2000.) ;;
# let g = gtcspp r 0.8 0.85 0.5 120. 10 ;;
val g : int list list * float * float =
  ([[1; 2; 3; 4; 5; 6; 7; 8; 9; 10]; [11; 12; 13; 14; 15; 16; 17];
    [18; 19; 20; 21; 22]; [23; 24; 25; 26; 27]; [28; 29; 30; 31; 32];
    [33; 34; 35; 36]; [37; 38; 39; 40]; [41; 42; 43; 44]; [45; 46; 47; 48];
    [49; 50; 51; 52]; [53; 54; 55]; [56; 57; 58]; [59; 60; 61]; [62; 63; 64];
    [65; 66; 67]; [68; 69; 70]; [71; 72; 73]; [74; 75; 76]; [77; 78; 79];
    [80; 81; 82]; [83; 84; 85]; [86; 87; 88]; [89; 90; 91]; [92; 93; 94];
    [95; 96; 97]; [98; 99; 100]; [101; 102; 103]; [104; 105; 106];
    [107; 108]; [109; 110]; [111; 112]; [113; 114]; [115; 116]; [117; 118];
    [119; 120]; [121; 122]; [123; 124]; [125; 126]; [127; 128]; [129; 130];
    [131; 132]; [133; 134]; [135; 136]; [137; 138]; [139; 140]; [141; 142];
```

```
    [143; 144]; [145; 146]; [147; 148]; [149; 150]; [151; 152]; [153; 154];
    [155; 156]; [157; 158]; [159; 160]; [161; 162]; [163; 164]; [165; 166];
    [167; 168]; [169; 170]; [171; 172]; [173; 174]; [175; 176]; [177; 178];
    [179; 180]; [181; 182]; [183; 184]; [185; 186]; [187; 188]; [189; 190];
    [191; 192]; [193; 194]; [195; 196]; [197; 198]; [199; 200]],
   4.56015043370027406, 119.882005782670291)
# let p = ... (* partition ci-dessus *)
# simulation r p 0.8 0.85 0.5 1000 ;;
- : float * float = (4.5115, 120.132)
# simulation r p 0.8 0.85 0.5 10000 ;;
- : float * float = (4.5937, 119.7342)
# simulation r p 0.8 0.85 0.5 10000 ;;
- : float * float = (4.57585, 119.6841)
```

Encore une fois, on a bien des résultats cohérents. Cependant, les exemples ne font pas preuves pour le cas général. On admet la correction.

7.3.4. Efficacité et complexité de l'algorithme

Il est difficile d'établir une complexité temporelle explicite de cet algorithme.

On peut cependant remarquer que le coût temporel est majoré par $K \cdot 4^N$ où K est une constante positive. En effet, la fonction `extend_and_dominance` est de l'ordre en complexité temporelle de la longueur de la liste en paramètre. Comme les ensembles Λ_k sont majorés par 2^{k-2} pour $k \geqslant 2$ (car ce sont des ensembles d'étiquettes représentant des chemins entre 1 et k, eux mêmes représentant des partitions ordonnées de $[\![1; k-1]\!]$), la complexité temporelle de la double boucle dans la fonction `gtcspp` est majorée à une constante multiplicative près par :

$$\begin{aligned}
\sum_{j=2}^{N+1}\sum_{i=1}^{j-1}|\Lambda_i|\cdot|\Lambda_j| &\leqslant \sum_{j=2}^{N+1}\sum_{i=2}^{j-1} 2^{i-2}\,2^{j-2} \\
&\leqslant \sum_{j=2}^{N+1} 2^{j-2}(2^{j-2}-1) \\
&\leqslant \sum_{j=2}^{N+1} 2^{2(j-2)} \\
&\leqslant \sum_{j=0}^{N-1} 4^j \\
&\leqslant \frac{4^N-1}{4-1} \\
\sum_{j=2}^{N+1}\sum_{i=1}^{j-1}|\Lambda_i|\cdot|\Lambda_j| &\leqslant \frac{4^N}{3}.
\end{aligned}$$

Certes c'est un majorant en exponentiel, mais cele reste beaucoup mieux que B_N qui lui est plus grand que l'exponentielle en ordre de grandeur (on peut le voir avec les premières valeurs de B_N) On voit ainsi pourquoi cet algorithme est meilleur.

Pour se donner une idée, calculons le temps d'exécution du code ci-dessus. On utilise le programme suivant, où on fait varier `nPop` :

```
let nf = Float.of_int nPop in
let f_init x =
  let xf = Float.of_int x in
  (1. +. xf) /. (nf +. 1.)
in
let r = Array.init nPop f_init in
let t = Sys.time () in
let _ = gtcspp r 0.9 0.8 0.5 nf nPop in
Sys.time () -. t ;;
```

N	temps d'exécution
50	0,14 s
75	0,85 s
100	2,9 s
125	8,3 s
150	19 s
175	37 s
200	1 min 8 s
225	1 min 49 s
250	2 min 58 s

Remarque. Plus les probabilités de `r` sont faibles, plus l'algorithme est rapide (c'est difficile à expliquer).

Plus il y a de contraintes (plus B et l sont faibles), moins l'ensemble A est grand, et plus l'algorithme est rapide. Ci-dessus, on a pris le pire des cas au niveau des contraintes.

Les nombres Se, Sp et λ ont aussi une influence.

On arrive à calculer des résultats pour N beaucoup plus grand que 13 en un temps raisonnable.

Quatrième partie
Applications

8. Dépistage du Covid-19 par test RT-PCR (prélèvement naso-pharyngé) dans les laboratoires français métropolitains

On va appliquer le principe du test groupé à l'échelle d'un laboratoire français de la Métropole réalisant des tests RT-PCR par prélèvement naso-pharyngé, sur une période d'un jour. On supposera pour simplifier que les tests de l'ensemble d'une journée sont réalisés en une seule série.

8.1. Détermination du nombre de personnes testés en une journée pour un laboratoire

D'après la base de données recençant les lieux de dépistage du Covid-19 (version du 5 avril 2022), que l'on trouve sur le site https://www.data.gouv.fr/fr/datasets/lieux-de-depistage-covid-19-laboratoires-et-pharmacies-sante-fr/, on compte 3 978 (sur 4 211) laboratoires réalisant ce type de test.

Dans la suite, on va se fixer une semaine particulère pour récupérer des données relatives au dépistage. On choisit de considérer la semaine du 3 au 9 janvier 2022. Les données proviennent du site https://drees.solidarites-sante.gouv.fr/delais-covid19-2022-01-13.

Sur cette semaine ont été réalisés 11 966 000 tests (RT-PCR naso-pharyngé et salivaire + antigénique). Parmi ceux-ci, il y a eu 67,0 % de tests antigéniques donc 33,0 % de tests RT-PCR (naso-pharyngé et salivaire). Cela fait 3 948 780 tests.

On compte toujours d'après le même site environ 1 040 509 tests salivaires, ce qui fait 2 908 271 tests RT-PCR naso-pharyngé.

Le nombre de personnes testés en une journée dans un laboratoire est ainsi en moyenne $\frac{2\,908\,271}{3\,978} \cdot \frac{1}{7} \approx \boxed{104}$. Cela correspond à notre N.

8.2. Détermination de la population testée

En fait, les tests RT-PCR salivaires sont majoritairement réalisés pour les individus d'âge faible (0 – 15 ans). On va éviter de considérer ces individus.

On va approximer notre population à considérer grâce aux proportions des classes d'âge de la population française métropolitaine au 1er janvier 2020 (données les plus récentes disponibles, issues de https://www.insee.fr/fr/statistiques/1892088?sommaire=1912926). Dans le tableau qui suit, les proportions corrigées correspondent aux proportions dont on a ajouté équitablement celle de la classe d'âge 0 – 9 ans et la moitié de celle des 10 – 19 ans : c'est pour prendre en compte la remarque ci-dessus au sujet des tests salivaires. Le nombre d'individus pour chaque classe d'âge est calculé en multipliant 104 par la proportion corrigée.

classe d'âge	proportion dans la population française métropolitaine	proportion corrigée	nombre d'individus considérés
0 – 9 ans	0,114	0	0
10 – 19 ans	0,123	0,061 5	6
20 – 29 ans	0,111	0,133	14
30 – 39 ans	0,129	0,151	16
40 – 49 ans	0,128	0,150	16
50 – 59 ans	0,131	0,153	16
60 – 69 ans	0,120	0,142	15
70 – 79 ans	0,086	0,108	11
80 – 89 ans	0,049	0,071	7
90 ans et plus	0,014	0,036	4
total	1,005	1,005 5	105

8.3. Détermination des probabilités d'être malade

On assimile la probabilité d'être malade au taux de prévalence sur un jour : c'est la proportion de malades sur un jour dans la population.

On dispose des données suivantes : les taux d'incidence sur une semaine pour les classes d'âge du tableau ci-dessus. Le taux d'incidence sur une semaine est la proportion de nouveaux malades sur la semaine dans la population. Ainsi, le taux de prévalence sur une semaine est supérieur à celui de l'incidence sur cette même période.

Voici les données, issues cette fois-ci de https://geodes.santepubliquefrance.fr/#c=indicator&f=90&i=sp_pe_tb_heb.tx_pe_hebdo&s=2022-S01&t=a01&view=map2.

classe d'âge	taux d'incidence sur la semaine du 3 janvier 2022 (moyenne sur l'ensemble des départements métropolitains)
0 – 9 ans	0,025 231
10 – 19 ans	0,044 670
20 – 29 ans	0,051 147
30 – 39 ans	0,036 780
40 – 49 ans	0,029 607
50 – 59 ans	0,020 310
60 – 69 ans	0,011 259
70 – 79 ans	0,008 167
80 – 89 ans	0,006 684
90 ans et plus	0,008 636

Les taux de prévalence P et d'incidence I sur une semiane sont reliées par l'équation :

$$\frac{nP}{1-nP} = I \times d$$

où n est la population totale et d la durée de la maladie. Ici, on peut prendre $d \approx 2$ semaines (donc $d = 1\,209\,600\,\mathrm{s}$).

Ainsi, on a :

$$nP = dI - ndPI \qquad \text{donc} \qquad nP(1 + dI) = dI \qquad \text{donc} \qquad P = \frac{dI}{n(1 + dI)}.$$

On donne un exemple de calcul du taux d'incidence pour la deuxième classe. La population totale française métropolitaine est $N_{\text{total}} = 64\,897\,954$. D'après le tableau des proportions des classes d'âge, on a donc $n = 0{,}123 N_{\text{total}} = 7\,982\,448{,}342$. D'où :

$$P_{10\,-\,19} = \frac{d \times 0{,}044\,670}{7\,982\,448{,}342 \times (1 + d \times 0{,}044\,670)} = 1{,}252\,725\,292\,082\,337 \cdot 10^{-07}.$$

où d est exprimé en secondes.

On en déduit le taux par jour en multipliant par $24 \times 60 \times 60$ le nombre de secondes dans un jour.

On répète ce raisonnement pour les autres classes, et on obtient :

classe d'âge	taux de prévalence sur un jour
0 – 9 ans	0,011 677 870 529 753 075
10 – 19 ans	0,010 823 546 523 591 392
20 – 29 ans	0,011 993 687 769 973 178
30 – 39 ans	0,010 320 084 783 466 292
40 – 49 ans	0,010 400 653 808 372 587
50 – 59 ans	0,010 162 341 005 939 044
60 – 69 ans	0,011 093 525 939 553 602
70 – 79 ans	0,015 478 908 247 628 587
80 – 89 ans	0,027 166 453 371 408 12
90 ans et plus	0,095 085 244 759 531 6

On a besoin d'avoir les probabilités triées dans l'ordre. On réarrange le tableau et on en déduit la composition exacte de notre population :

classe d'âge	nombre d'individus	taux de prévalence sur un jour
50 – 59 ans	16	0,010 162 341 005 939 044
30 – 39 ans	16	0,010 320 084 783 466 292
40 – 49 ans	16	0,010 400 653 808 372 587
10 – 19 ans	6	0,010 823 546 523 591 392
60 – 69 ans	15	0,011 093 525 939 553 602
0 – 9 ans	0	0,011 677 870 529 753 075
20 – 29 ans	14	0,011 993 687 769 973 178
70 – 79 ans	11	0,015 478 908 247 628 587
80 – 89 ans	7	0,027 166 453 371 408 12
90 ans et plus	4	0,095 085 244 759 531 6

classe	numéros pour la partition
50 – 59 ans	1 à 16
30 – 39 ans	17 à 32
40 – 49 ans	33 à 48
10 – 19 ans	49 à 54
60 – 69 ans	55 à 69
20 – 29 ans	70 à 83
70 – 79 ans	84 à 94
80 – 89 ans	95 à 101
90 ans et plus	102 à 105

8.4. Application de l'algorithme

D'après le site https://solidarites-sante.gouv.fr/IMG/pdf/annexe_2_-_tableau_des_tests_disponibles_et_leurs_indication1.pdf?TSPD_101_R0=087dc22938ab2000901648b20f385ffa9a9cbc6b93d2718065a6ce8113b61b51323e42739a0d70ab0858242c321430009f4289876448fcbddedcede23254e6f5e156cab5d0236ac351c4be1a3e1fde3e79a8856381026665da5b4b76f02678ec (date du 13 octobre 2021), le test peut être assimilé à un test parfait : $Se = Sp = 1$.

La population est hétérogène ; on applique donc l'algorithme décrit dans la sous-section 7.3.

Comme le test est parfait, on choisit B trop grand car le programme va chercher à minimiser l'espérance du nombre de tests (on prend $B = 105$). À vrai dire, on peut prendre n'importe quel $B \geqslant 0$ car toutes les espérances de faux négatifs et positifs impliquées sont nulles. C'est pourquoi le choix de λ n'importe pas (on prend ici $\lambda = 0{,}5$). Par ailleurs, d'après [4], prendre $l = 32$ est suffisant pour détecter un échantillon infecté du virus dans un mélange d'échantillons majoritairement sains.

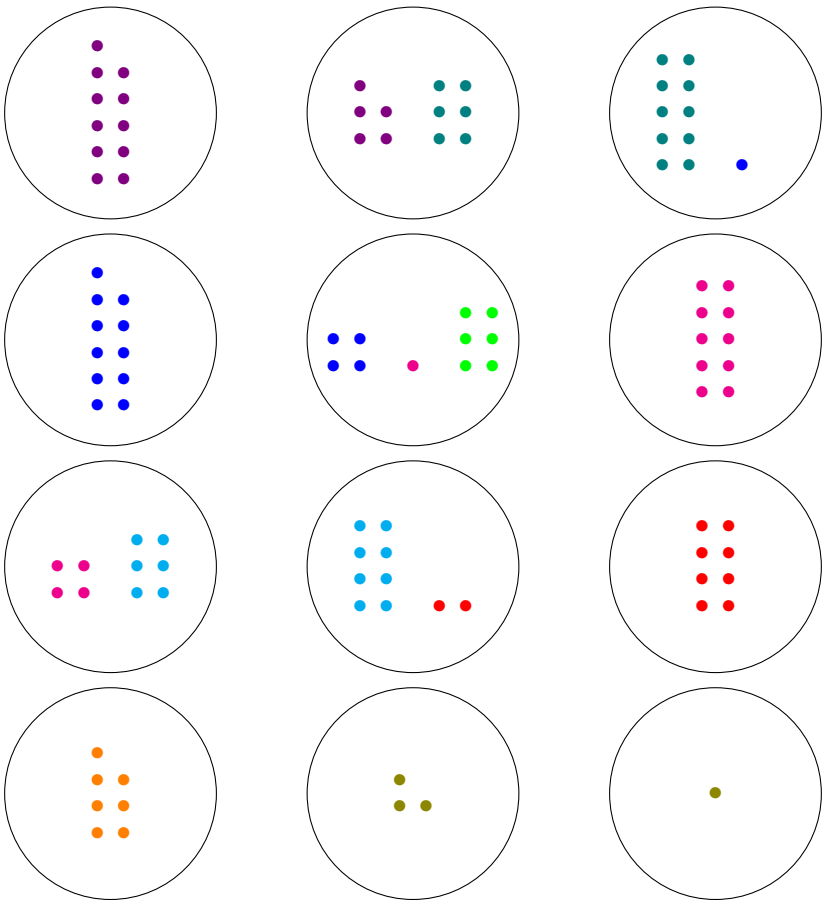
Voici le résultat :

```
# let r5 = Array.make 16 0.010162341005939044 in
let r3 = Array.make 16 0.010320084783466292 in
let r4 = Array.make 16 0.010400653808372587 in
let r1 = Array.make 6 0.010823546523591392 in
let r6 = Array.make 15 0.011093525939553602 in
let r2 = Array.make 14 0.011993687769973178 in
let r7 = Array.make 11 0.015478908247628587 in
let r8 = Array.make 7 0.02716645337140812 in
let r9 = Array.make 4 0.0950852447595316 in
let r = Array.concat [r5; r3; r4; r1; r6; r2; r7; r8; r9] in
gtcspp r 1. 1. 0.5 105. 32;;
- : int list list * float * float =
([[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11];
  [12; 13; 14; 15; 16; 17; 18; 19; 20; 21; 22];
  [23; 24; 25; 26; 27; 28; 29; 30; 31; 32; 33];
```

```
  [34; 35; 36; 37; 38; 39; 40; 41; 42; 43; 44];
  [45; 46; 47; 48; 49; 50; 51; 52; 53; 54; 55];
  [56; 57; 58; 59; 60; 61; 62; 63; 64; 65];
  [66; 67; 68; 69; 70; 71; 72; 73; 74; 75];
  [76; 77; 78; 79; 80; 81; 82; 83; 84; 85];
  [86; 87; 88; 89; 90; 91; 92; 93; 94]; [95; 96; 97; 98; 99; 100; 101];
  [102; 103; 104]; [105]],
 0., 24.4988436648694616)
```

On obtient 12 groupes avec une espérance du nombre de tests égale environ à 25,5 : c'est très efficace (c'est normal, le test est parfait).

Pour représenter la partition obtenue, un schéma sera peut-être plus clair. Les cercles représentent les groupes et les points les individus :



classe d'âge	couleur
10 – 19 ans	
20 – 29 ans	
30 – 39 ans	
40 – 49 ans	
50 – 59 ans	
60 – 69 ans	
70 – 79 ans	
80 – 89 ans	
90 ans et plus	

8.5. Cas où le test n'est (légèrement) pas parfait

On va regarder ce que nous donne l'algorithme si à la place de prendre $Se = Sp = 1$, on prend une sensibilité et une spécificité proches de 1. Prenons $Se = Sp = 0{,}99$.

```
# let r5 = Array.make 16 0.010162341005939044 in
let r3 = Array.make 16 0.010320084783466292 in
let r4 = Array.make 16 0.010400653808372587 in
let r1 = Array.make 6 0.010823546523591392 in
let r6 = Array.make 15 0.011093525939553602 in
let r2 = Array.make 14 0.011993687769973178 in
let r7 = Array.make 11 0.015478908247628587 in
let r8 = Array.make 7 0.02716645337140812 in
let r9 = Array.make 4 0.0950852447595316 in
let r = Array.concat [r5; r3; r4; r1; r6; r2; r7; r8; r9] in
gtcspp r 0.99 0.99 0.5 105. 32;;
- : int list list * float * float =
([[1; 2]; [3; 4]; [5; 6]; [7; 8; 9]; [10; 11]; [12; 13]; [14; 15]; [16; 17];
  [18; 19]; [20; 21]; [22; 23]; [24; 25]; [26; 27]; [28; 29]; [30; 31];
  [32; 33]; [34; 35]; [36; 37]; [38; 39]; [40; 41]; [42; 43]; [44; 45];
  [46; 47]; [48; 49]; [50; 51]; [52; 53]; [54; 55]; [56; 57]; [58; 59];
  [60; 61]; [62; 63]; [64; 65]; [66; 67]; [68; 69]; [70; 71]; [72; 73];
  [74; 75]; [76; 77]; [78; 79]; [80; 81]; [82; 83]; [84; 85]; [86; 87];
  [88; 89]; [90; 91]; [92; 93]; [94; 95]; [96; 97]; [98; 99]; [100; 101];
  [102; 103]; [104; 105]],
 0.0293198719679262573, 56.2298151796444117)
```

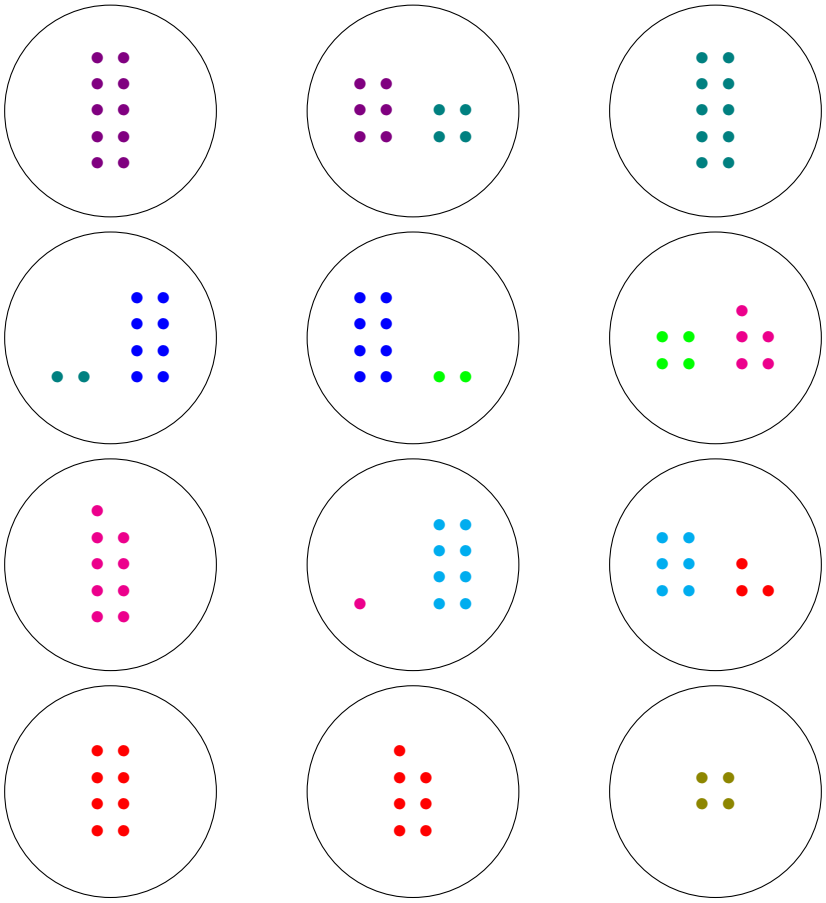
Cette fois-ci, on n'obtient que des groupes de 2, à l'exception du 4e groupe composé de 3 individus entre 50 et 59 ans. Ainsi, les groupes sont formés à une exception près de 2 individus de même classe d'âge ou de 2 individus de classe différentes (les groupes `[16; 17]`, `[32; 33]`, `[48; 49]`, `[54; 55]` et `[94; 95]`).

On remarque que l'espérance du nombre de tests est plus élevé que précédemment (un peu plus que le double). On remarque cependant que le coût C est très faible : essayons

de diminuer la valeur de B au maximum, quitte à augmenter la valeur de C. De proche en proche, on peut diminuer B jusqu'à 25 ; après, la partition optimale n'existe pas.

```
# let r5 = Array.make 16 0.010162341005939044 in
let r3 = Array.make 16 0.010320084783466292 in
let r4 = Array.make 16 0.010400653808372587 in
let r1 = Array.make 6 0.010823546523591392 in
let r6 = Array.make 15 0.011093525939553602 in
let r2 = Array.make 14 0.011993687769973178 in
let r7 = Array.make 11 0.015478908247628587 in
let r8 = Array.make 7 0.02716645337140812 in
let r9 = Array.make 4 0.0950852447595316 in
let r = Array.concat [r5; r3; r4; r1; r6; r2; r7; r8; r9] in
gtcspp r 0.99 0.99 0.5 25. 32;;
- : int list list * float * float =
([[1; 2; 3; 4; 5; 6; 7; 8; 9; 10]; [11; 12; 13; 14; 15; 16; 17; 18; 19; 20];
  [21; 22; 23; 24; 25; 26; 27; 28; 29; 30];
  [31; 32; 33; 34; 35; 36; 37; 38; 39; 40];
  [41; 42; 43; 44; 45; 46; 47; 48; 49; 50];
  [51; 52; 53; 54; 55; 56; 57; 58; 59]; [60; 61; 62; 63; 64; 65; 66; 67; 68];
  [69; 70; 71; 72; 73; 74; 75; 76; 77]; [78; 79; 80; 81; 82; 83; 84; 85; 86];
  [87; 88; 89; 90; 91; 92; 93; 94]; [95; 96; 97; 98; 99; 100; 101];
  [102; 103; 104; 105]],
 0.072462357863931709, 24.8583123588454917)
```

On retrouve à peu près l'espérance du nombre de tests que pour le test parfait. La conception des groupes n'est cependant pas la même : on obtient 12 groupes différents, selon le schéma suivant :



9. Dépistage du VIH dans les laboratoires en Île-de-France

Appliquons cette fois-ci le principe du group testing à l'échelle d'un laboratoire en Île-de-France réalisant des tests sérologiques de dépistage du VIH, sur une période d'une semaine. Encore une fois, pour simplifier, on supposera que les tests de l'ensemble d'une semaine sont réalisés en une seule série.

9.1. Nombre de tests par semaine dans un laboratoire, et probabilité d'être malade

D'après le site https://www.data.gouv.fr/fr/datasets/laboratoires-de-biologie-medicale-idf/ (dernière mise à jour le 4 septembre 2014), il y a 823 laboratoires en Île-de-France.

D'après le site https://geodes.santepubliquefrance.fr/#bbox=201639,6288933,151641,111955&c=indicator&i=labovih.labovih_test&s=2019&t=a01&view=map2, en moyenne, 1 350 890 tests ont été réalisés en 2014. Cela fait donc environ 1 641 tests dans l'année pour un laboratoire. Sur une semaine, cela fait 37 tests sur une semaine. Ce dernier nombre correspond à notre N.

Toujours d'après le même site, en moyenne, 3,9 tests sérologiques sur 1 000 sont positifs, pour l'année 2014. On approxime la probabilité d'être malade pour un quelconque individu à ce nombre : $p = 0{,}003\,9$. On considère ainsi que notre population est homogène (on ne peut trouver des données plus précises).

Il nous reste juste à trouver la sensibilité et spécificité de ce test. D'après https://msmgf.org/files/msmgf/documents/TechBulletins/FR/MSMGF_Diagnostic.pdf, on prend $Se = 0{,}89$ et $Sp = 1$.

9.2. Application de l'algorithme

Le test n'est pas parfait. On décide dans un premier temps de ne pas se fixer une limite B (on prend $B = 37$). On applique l'algorithme de la sous-sous-section 5.3.3 avec $\lambda = 1$ car il n'y a pas de faux positifs ($Sp = 1$). L'algorithme minimise donc $\lambda\mathbb{E}(FN(\tau)) + (1-\lambda)\mathbb{E}(FP(\tau)) = \mathbb{E}(FN(\tau))$, *i.e.* l'espérance du nombre de faux négatifs.

```
# conceptionOptEff 37 0.0039 0.89 1. 1. 37. ;;
- : int list * float * float =
([1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1],
 0.0158729999999999982, 37.)
```

La conception idéale correspond au cas « classique ». L'espérance de faux négatifs est très faible. On va donc essayer de diminuer B pour optimiser le nombre de tests.

De proche en proche, on trouve que la valeur de B minimale au dessous de laquelle la partition optimale n'existe pas est $B = 4{,}288$.

```
# conceptionOptEff 37 0.0039 0.89 1. 1. 4.5 ;;
- : int list * float * float =
([18; 19], 0.0299999699999999939, 4.29809420974458778)
# conceptionOptEff 37 0.0039 0.89 1. 1. 4.298 ;;
Exception:
Failure "La partition optimale pour des groupes quelconques n'existe pas".
```

La conception optimale donne toujours une espérance de faux négatifs très faible : on peut prendre cette partition pour optimiser le dépistage du VIH. On prend $\tau = (18, 19)$, c'est-à-dire qu'on construit deux groupes de 18 et de 19 échantillons, pour une espérance du nombre de tests d'environ $4{,}3 \ll 37$.

Références

[1] Robert Dorfman. « The Detection of Defective Members of Large Populations ». In : *The Annals of Mathematical Statistics* 14.4 (déc. 1943), p. 436-440. DOI : 10.1214/aoms/1177731363. URL : https://doi.org/10.1214%2Faoms%2F1177731363.

[2] Matthew Aldridge, Oliver Johnson et Jonathan Scarlett. « Group Testing : An Information Theory Perspective ». In : *Foundations and Trends® in Communications and Information Theory* 15.3-4 (2019), p. 196-392. doi : 10.1561/0100000099. url : https://doi.org/10.1561%2F0100000099.

[3] Hrayer Aprahamian, Douglas R. Bish et Ebru K. Bish. « Optimal Risk-Based Group Testing ». In : *Management Science* 65.9 (sept. 2019), p. 4365-4384. doi : 10.1287/mnsc.2018.3138. url : https://doi.org/10.1287%2Fmnsc.2018.3138.

[4] Tifaout Almeftah et al. *Group design in group testing for COVID-19 : A French case-study*. Rapport de recherche. INRIA - Centre Lille Nord Europe, nov. 2020. url : https://hal.inria.fr/hal-03000715.

[5] Christopher R. Bilder. *Group testing for SARS-CoV-2 detection during the COVID-19 pandemic - Seminar given for Stanford U.* https://www.youtube.com/watch?v=w7pb86mb02Q. Juill. 2020.

A. Liste des codes sources