Costa da Quinta, João Filipe

TP Algorithme – Bonus 3

1.1)

Cf fichier python.

1.2)

La complexité de l'algorithme backtracking théorique est O(n^n), car on a pour la première reine n options, pour la deuxième aussi et si de suite jusqu'à la n-ieme reine.

En pratique on met en place des contraintes explicites, qui réduisent la complexité en O(n !) car pour la première reine on a n choix, pour la deuxième n-1 choix etc. Finalement on a la dernière condition implicite qui réduit le nombre de calculs, si on compare cette méthode au parcours d'un arbre, alors la condition implicite, fait en sorte qu'on ait pas besoin de parcourir en entier chaque branche.

On observe une complexité exp(x). J'ai utilisé le module Time comme pour le dernier TP, pour calculer le temps de calcul de solve(4) jusqu'à solve(14).

J'ai trouvé les résultats suivants :

N	Temps(s)	Nombre de solutions
4	0,0001513	2
5	0,0004909	10
6	0,0021659	4
7	0,0067284	40
8	0,0351407	92
9	0,1783826	352
10	0,8316256	724
11	4,4153123	2680
12	28,218636	14200
13	161,92703	73712
14	1.073	365596

Tableau 1

X	x^2	exp(x)
4	16	54,5
5	25	148,4
6	36	403,4
7	49	1096,6
8	64	2980,9
9	81	8103,08
10	100	22026,4
11	121	59874,1
12	144	162754,7
13	169	442413,3
14	196	1202604,2

Tableau 2

(Ceci sont des résultats pour ma machine personnelle)

Tableau 1 : solve(N), a trouvé (Nombre de solutions) Solutions, en (Temps(s)) secondes.

Tableau 2 : Croissance de la fonction x^2 et exp(x) - (pour comparaison).

Temps(s) pour n(14) = 1073 secondes, et temps(s) pour n(13) = 161,9 secondes. 1073/161,9 = 6,63, alors que exp(14)/exp(13) = 2,72.

Pour N=6, le nombre de solutions est plus petit que pour N=5, mais il est quand même plus long, nous remarquons que le temps de calcul augmente considérablement avec le choix de N, ainsi que le nombre de solutions, et plus de solutions implique plus de branches entièrement parcourues. Augmentation de solutions implique directement une augmentation dans le temps de calcul.

1.3)

La méthode Greedy marche différemment, en placent aléatoirement et déplaçons par la suite les reines qui sont mal placés a donc une complexité différente, O(k*n^2) ou k est le nombre de reines déplaces, si on déplace toutes les reines, on a O(n^3) c'est le pire des cas.

Greedy est donc moins complexe (cf tableau exo 1.2)) que Backtracking, mais Backtracking a aussi ses points forts, il assure un parcours total de toutes les solutions, et en plus, on a les solutions dans un ordre logique.

Greedy est plus rapide pour trouver une solution, mais pour les trouver toutes, on préfère Backtracking.

1.4)

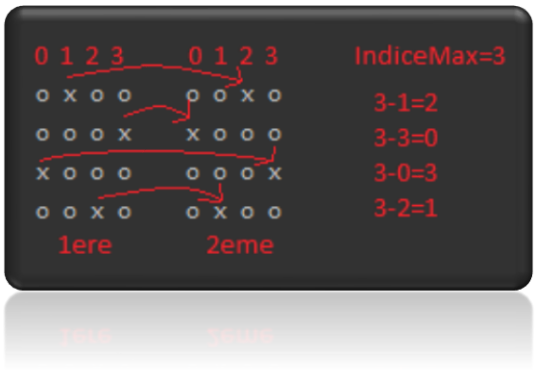
Solution N=4:

Le nombre de solutions pour tout N est pair, ceci est du au fait que toute solution a une solution « miroir ». Voici la solution pour N=4, il existe 2 solutions, [[1,3,0,2],[2,0,3,1]], (voir image ci prés).

Soit N=4, la 1ere colonne est représenté par l'indice [0], la 2eme par l'indice [1], 3eme par [2], la 4eme et dernière par l'indice [3]. Soit solution1, la première solution, et solution2 sa solution miroir, solution1 = [1,3,0,2]

- IndiceMax = 3 = N-1. Solution1[0]=1.
- i=0, et on l'incrémente avec la boucle while i<len(Solution1)
- Mettons : IndiceMax – Solution[i] = Solution2[i]
- On trouvera Solution2 = [3-1,3-3,3-0,3-2] = [2,0,3,1] qui est bien la 2eme solution pour N=4.

Schéma miroir N=4:



Ceci est vrait pour tout N pair.

N=4 c'est simple, car il existe que 2 solutions, mais pour N=6 nous avons 4 solutions, la première est miroir de la dernière (4eme), la 2eme de l'avant dernière (3eme), ceci est très intéressant, car ca implique qu'on peut réduire par moitié le temps de calcul pour tout N pair, pour savoir si on a la moitié des solutions, il suffit de faire que la première moitié des solutions, pour N=6, on fait que les solutions ou l'indice [0] = [0,1,2], pour [0] = [3,4,5] c'est en fait les miroirs.

Pour N impair, on peut faire presque la même chose, mettons N=7, alors on calcule les solutions l'indice [0] = [0,1,2,3], ensuite les solutions de indice [0]=[4,5,6], sont miroir de l'indice [0] = [0,1,2].

- Pour N pair, on a divisé le temps de calcul par 2.
- Pour N impair on a que le temps de calcul est :
  t(s)-temps(s)[i-> milieu] divisé par 2, + temps(s)[i->milieu].

(temps(s)[i->milieu]=temps de calcul des solutions indice i = milieu (pour N=7, i=3) en seconds)

(arrondissons à temps(N)/2)

Il existe aussi des solutions qui sont verticalement symétriques ainsi que dans la diagonale.

Finalement si on a toutes les solutions pour N, il suffit de trouver une qui n'a pas de reine sur la diagonale principale, l'augmenter en taille N+1, et mettre la nouvelle reine dans la diagonale principale, ceci est vrai pour tout N.