INF8775 – Analyse et conception d’algorithmes

Rapport TP2 – Automne 2023

Nom, prénom, matricule des membres	FALICOFF, Maximiliano, 2013658 Houtart, Thomas, 1895556
Note finale / 30	0

Présentation des résultats

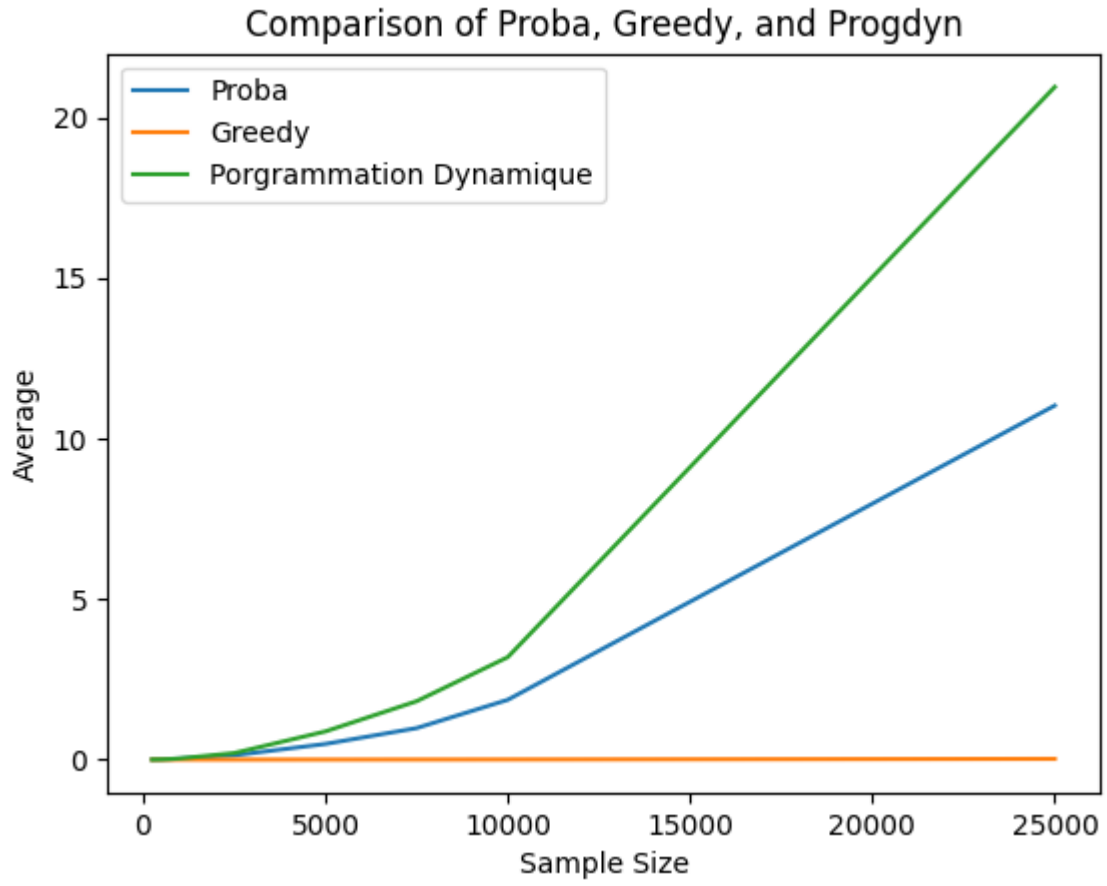
	/ 1,5pts

Tableau des résultats

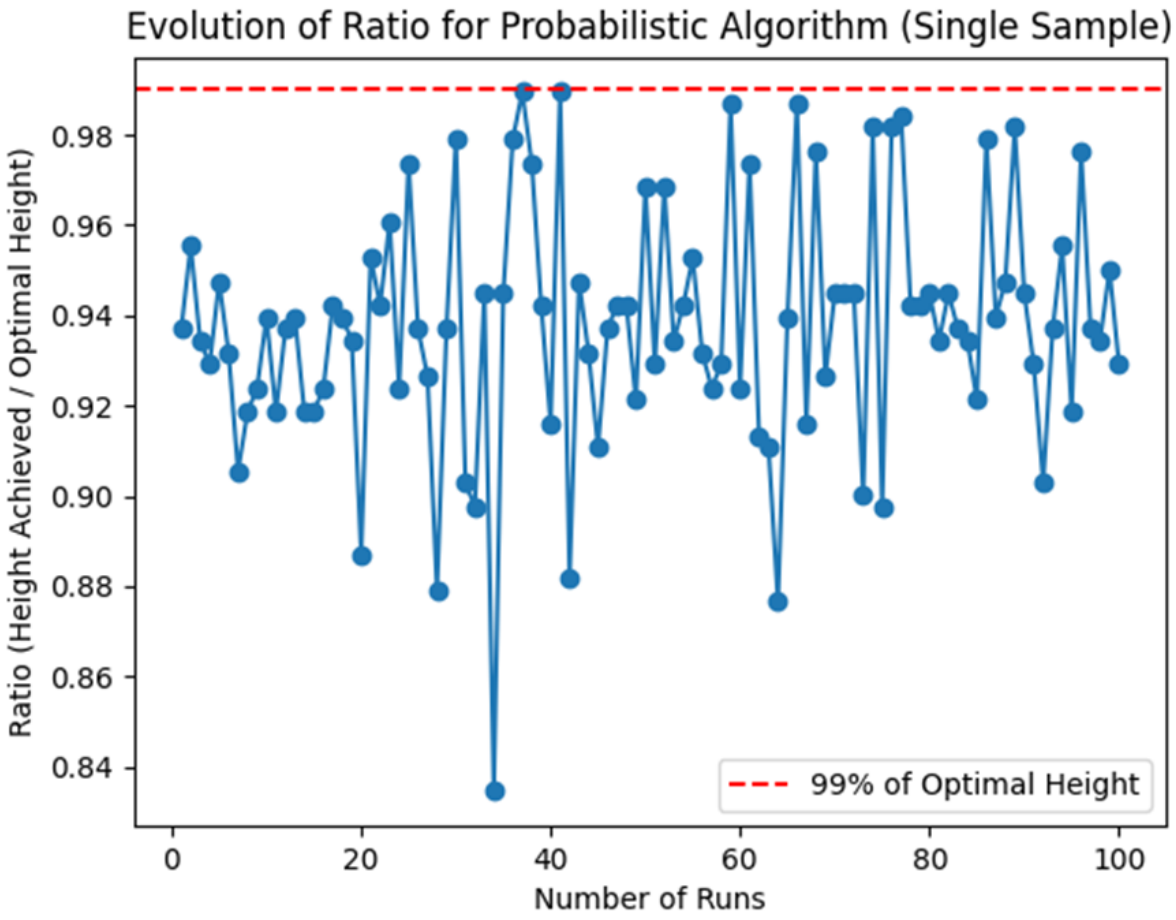
Tous les temps sont en secondes dans le tableau et dans les graphiques

Sample Size	Avg Time Proba	Avg Height Proba	Avg Time Greedy	Avg Height Greedy	Avg Time Progdyn	Avg Height Progdyn
250	0.004560351	38.9	0.000101852	42.1	0.002399278	43.9
500	0.009397626	54.4	0.000161076	58.6	0.007078767	59.8
750	0.018569636344909667	83.9	0.000258946	85.7	0.019647550582885743	92.4
1000	0.030108165740966798	104.8	0.000370097	105.3	0.036617612838745116	111.8
2500	0.1349479913711548	172.2	0.000923729	173.6	0.2047065019607544	186.7
5000	0.48916633129119874	294.9	0.002201676	294.2	0.8796133518218994	311.7
7500	0.9824604272842408	370.6	0.003616071	368.7	1.8201456785202026	389.1
10000	1.864341711997986	454.3	0.005463719	445.5	3.193472433	470.6
25000	11.032745385169983	858.1	0.024072027206420897	831.4	20.958458995819093	881.9

Note: Height représente la hauteur atteinte par nos algorithme



Précision en fonction du nombre de relances



Nous avons réalisé ce test avec un exemplaire de taille 7500. Nous avons lancé l'algorithme dyn pour avoir la hauteur optimale puis on a lancé proba 100 fois puis on a comparé la hauteur atteinte / hauteur optimale

Analyse et discussion

Faites une analyse asymptotique théorique du temps de calcul pour chaque algorithme.

	/ 7,5 pt

Afin de déterminer la complexité de chacun des algorithmes que nous utilisons, nous procéderons au calcul de la complexité pour chaque ligne. Nous poserons n comme étant le nombre d'éléments dans le tableau.

PS: Nous ignorons les try/catch dans cette analyse, car leur impact est négligeable sur la complexité (O(1)) et car leur but n'est pas relié directement au fonctionnement de l'algorithme.

PS: Nous avons refait un peu le code pour le rendre un peu plus lisible, mais pour garder l'analyse des lignes consistante nous avons laissé les images avec le code original.

Nous avons une classe commune Block qui représente un block

```
3 usages  Maximiliano Falicoff
class Block:
    Maximiliano Falicoff
    def __init__(self, height, width, depth, fitness=-1):
        self.width = width
        self.height = height
        self.depth = depth
        if fitness == -1:
            self.fitness = self.height / self.depth * self.width * self.height
        else:
            self.fitness = fitness

    Maximiliano Falicoff
    def __str__(self):
        return (f"height: {self.height}, width: {self.width}, depth: {self.depth}, "
                f"density: {self.height * self.width * self.depth}, "
                f"fitness: {self.fitness}")

    Maximiliano Falicoff
    def __eq__(self, other):
        return self.width == other.width and self.depth == other.depth and self.height == other.height

    Maximiliano Falicoff
    def __hash__(self):
        return hash((self.height, self.width, self.depth))

    1 usage (1 dynamic)  Maximiliano Falicoff
    def fits_on_other_block(self, previous_block):
        return self.width < previous_block.width and self.depth < previous_block.depth

```

Toutes le méthodes de cette classe sont en O(1) car elle ne réalisent que des opérations élémentaires

Glouton :

```
Maximiliano Falicoff
9   def return_keys(blocks):
10      return list(dict.fromkeys(blocks))
11
12
    1 usage  Maximiliano Falicoff
13  def does_block_fit_on_previous_block(new_block, previous_block):
14      return new_block.width < previous_block.width and new_block.depth < previous_block.depth
15
16
    4 usages  Maximiliano Falicoff *
17  def greedy(the_list, max_height, max_depth, max_width):
18      try:
19          sorted_blocks = sorted(list(the_list), key=lambda x: fitness(x, max_height, max_depth, max_width), reverse=True)
20
21          used_blocks = []
22          height = 0
23
24          while len(sorted_blocks) != 0:
25              if len(used_blocks) == 0:
26                  block = sorted_blocks.pop(0)
27                  used_blocks.append(block)
28                  height += block.height
29                  continue
30              block = sorted_blocks.pop(0)
31              if does_block_fit_on_previous_block(block, used_blocks[-1]):
32                  used_blocks.append(block)
33                  height += block.height
34
35          return used_blocks, height
36
37      except Exception:
38          print(traceback.format_exc())
39          # or
40          print(sys.exc_info()[2])
41
```

La ligne 19 fait un tri, soit une opération de complexité O(n log n). Des affectations de variable sont faites à la ligne 21 et 22 et ont chacune une complexité de O(1). La boucle while à la ligne 24 fait un nombre d'itération en pire cas et à donc une complexité de O(n). Chaque opérations dans la boucle for (ligne 25 à 33) ont une complexité de O(1). Le return à la ligne 25 est d'une complexité O(1). L'algorithme à donc deux groupes d'opérations, soit le sort d'une complexité O(n log n) et la boucle while de complexité O(n). Sachant que la complexité d'un algorithme est définie par son opération la plus coûteuse, la complexité de cet algorithme est de O(n log n).

Proba

```
5 usages (1 dynamic)  Maximiliano Falicoff
def fitness(block):
    block.fitness = (block.width * block.depth) + block.height
    return block.fitness


2 usages  Maximiliano Falicoff
def probability(blocks):
    try:
        for block in blocks:
            fitness(block)

        used_blocks = set()
        height = 0

        while len(blocks) > 0:
            fitness_values = [obj.fitness for obj in blocks]
            probabilities = softmax(fitness_values)
            chosen_index = numpy.random.choice(len(blocks), p=probabilities)
            chosen_block = list(blocks)[chosen_index]
            blocks.remove(chosen_block)
            chosen_block.fitness = 0

            if len(used_blocks) == 0:
                used_blocks.add(chosen_block)
                height += chosen_block.height
                continue

            for used_block in used_blocks:
                if not chosen_block.fits_on_other_block(used_block):
                    break
            else:
                used_blocks.add(chosen_block)
                height += chosen_block.height

        return sorted(used_blocks, key=lambda block: (block.width, block.depth), reverse=True), height
    except Exception as e:
        print(traceback.format_exc())
        # or
        print(sys.exc_info()[2])
```

La boucle à la ligne 12 à une complexité de O(n) et les opérations de la ligne 13, 6 et 7 ont chacune une complexité de O(1). Les affectation à la ligne 15 et 16 sont aussi d'une complexité O(1). Pour la boucle 18, en pire cas, nous avons n itérations, donc O(n). Pour toutes les opérations entre la ligne 19 et 36, à l'exception de la ligne 31, nous avons une complexité de O(n). Pour la ligne 31, la taille de début commence à 1 à la première itération de la boucle de la ligne 22 et croît de 1 après chaque itération. Le nombre d'itération pour cette boucle peut donc augmenter jusqu'à n opérations en pire cas, et à donc une complexité de O(n). La ligne 38 à une complexité de O(1). Sachant que nous avons une boucle d'une complexité de O(n/2) qui est dans une boucle de O(n), nous savons que la complexité du groupe de la ligne 26 à 36 est de O(n^2). Donc, l'algorithme est d'une complexité O(n^2).

Dynamique

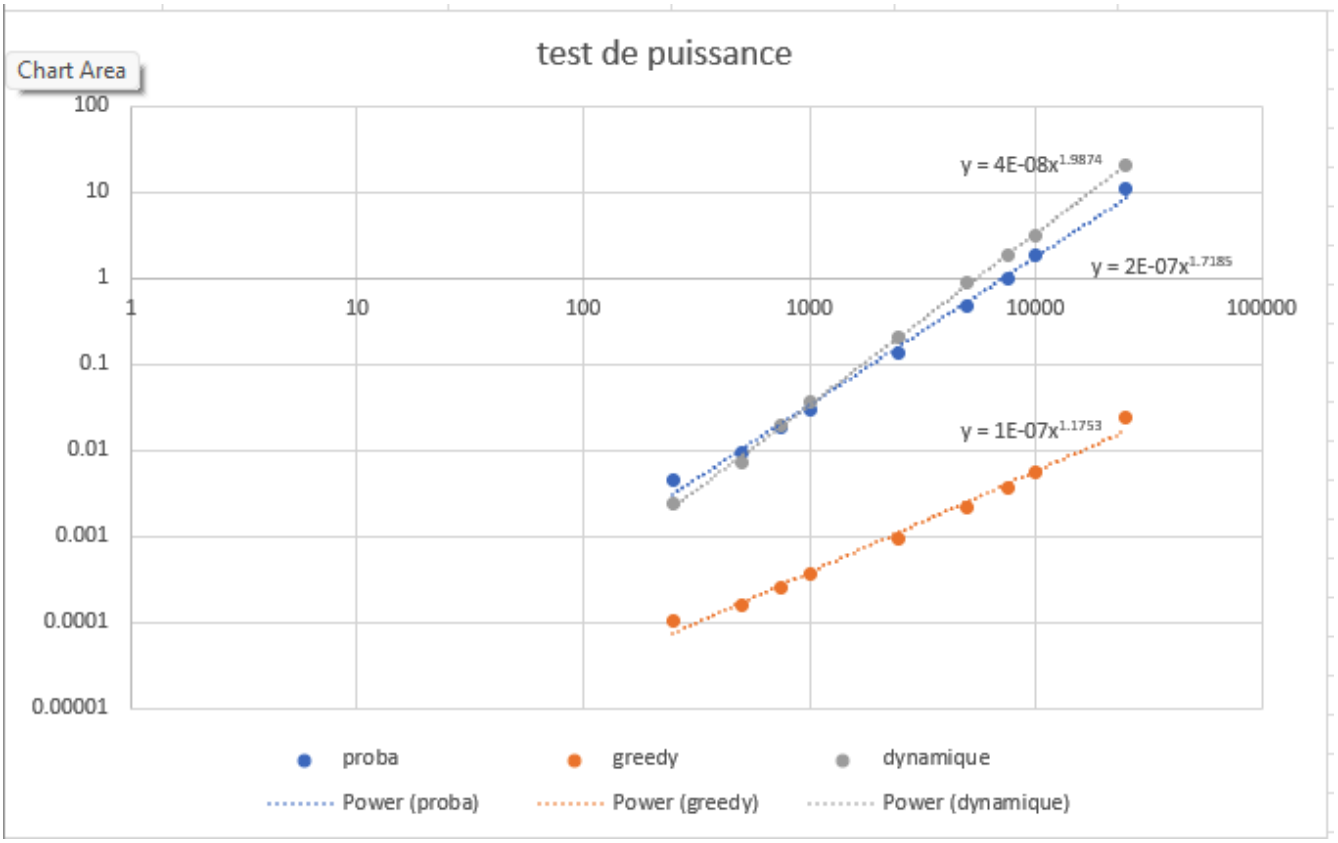
```
5 usages (1 dynamic)  Maximiliano Falicoff
def fitness(block):
    return block.depth * block.width


2 usages  Maximiliano Falicoff *
def dyn(the_list):
    try:

        sorted_blocks = sorted(list(the_list), key=lambda x: fitness(x), reverse=True)

        stacks_table = [()] * len(sorted_blocks)

        for i in range(len(sorted_blocks)):
            stacks_table[i] = (sorted_blocks[i].height, [sorted_blocks[i]])

        for i in range(1, len(sorted_blocks)):
            for j in range(0, i):
                if (sorted_blocks[i].depth < sorted_blocks[j].depth
                        and sorted_blocks[i].width < sorted_blocks[j].width
                        and stacks_table[i][0] < stacks_table[j][0] + sorted_blocks[i].height):
                    stacks_table[i] = (stacks_table[j][0] + sorted_blocks[i].height, stacks_table[j][1] + [sorted_blocks[i]])

        return max(stacks_table, key=itemgetter(0))[1], max(stacks_table, key=itemgetter(0))[0]

    except Exception:
        print(traceback.format_exc())
        # or
        print(sys.exc_info()[2])
```

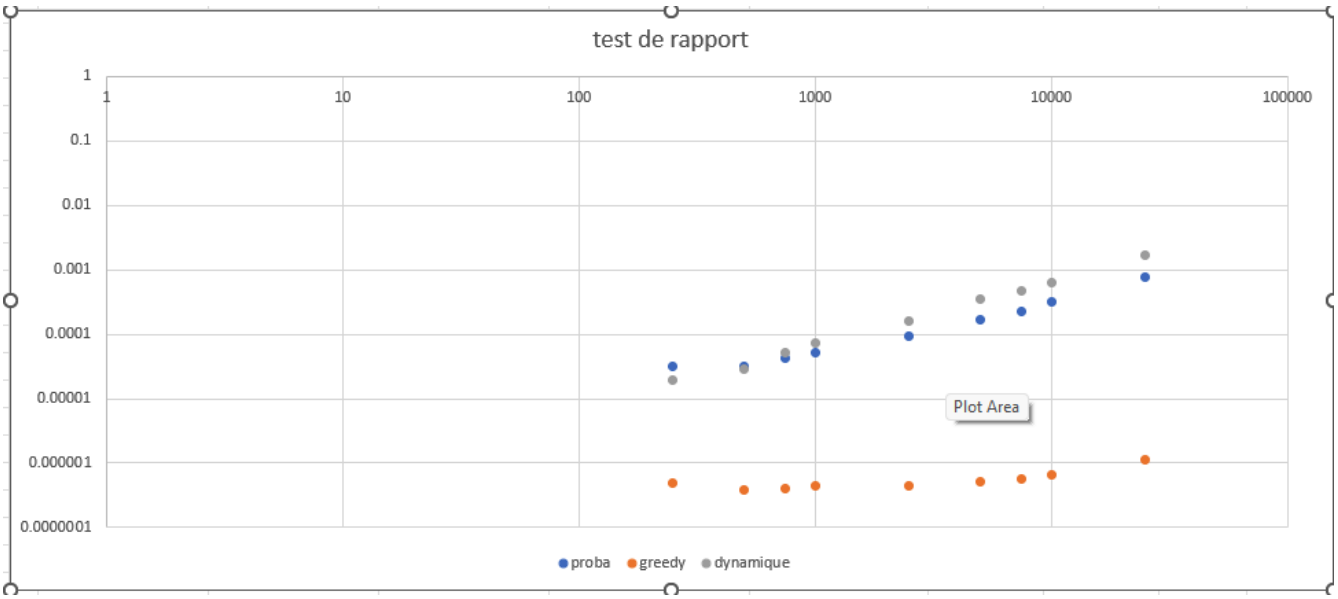
La ligne 15 fait un tri, soit une opération d'une complexité O (n log n). La ligne 17 est une affectation d'une complexité O(1). La boucle for à la ligne 18 est d'une complexité O(n). La ligne 19 est d'une complexité O(1). La boucle for à la ligne 20 est d'une complexité O(n). La boucle for à la ligne 21 commence a une opération et croît de 1 à chaque opération jusqu'à n. Sa complexité est donc aussi de O(n). Les opérations à la ligne 24, 25, 26 et 27 sont d'une complexité O(1). L'opération à la ligne 29 est aussi d'une complexité O(1). Sachant que nous sommes assuré de faire O(n) opération pour la boucle 19 et que la complexité du groupe de la ligne 22 à 27 et O(n^2) (car on a une boucle de complexité O(n) dans une boucle O(n), la complexité de cet algorithme est de O($n + n^2$).

Servez-vous de vos temps d'exécution pour confirmer et/ou préciser l'analyse asymptotique théorique de vos algorithmes avec la méthode hybride de votre choix.

	/ 7,5 pt

La méthode peut varier d'un algorithme à l'autre. Justifiez les choix ici et montrez vos graphiques .





Nous remarquons dans le tests de puissance que la croissance polynomiale pour l'algorithme greedy est beaucoup plus bas que l'algorithme probabiliste et dynamique, ce qui est normal sachant que c'est l'algorithme ayant la plus basse complexité. Nous remarquons aussi que l'algorithme dynamique croît légèrement plus vite que l'algo probabiliste, ce qui suit aussi les résultats de notre analyse de la complexité asymptomatique théorique.

Pour l'algorithme glouton, l'exposant que nous avons obtenu au test de puissance est environ 1.18, soit supérieur à 1. Ceci est normal sachant que la complexité théorique est équivalente à O(n log n).

Pour l'algorithme probabiliste, l'exposant que nous avons obtenu est environ 1.72. Il est normale que le résultats soit proche de deux et qu'il soit inférieur à 2, sachant que l'algorithme est borné supérieurement à n^2.

Pour l'algorithme dynamique, l'exposant que nous avons obtenu est environ 1.99. Il est un peu surprenant que le résultat ne soit pas supérieur à deux, sachant que la complexité théorique trouvée, soit O($n + n^2$), suggère que l'on obtient un résultat supérieur à 2.

Le test du rapport montre que nos résultats pour l'ensemble des algorithmes ne convergent pas. Cela signifie que les résultats obtenus au tests de puissances sont des sous-estimations et que les vrais résultats sont supérieurs. Ce résultat offre une explication plausible sur la raison pour laquelle le résultat obtenu au test de puissance pour l'algorithme dynamique est inférieur à 2.

Discutez des trois algorithmes en fonction de la qualité respective des solutions obtenues, de la consommation de ressources (temps de calcul, espace mémoire) et de la difficulté d'implantation.

Au niveau du temps de calcul, nous remarquons que l'algorithme glouton est nettement supérieur aux autres algorithmes, que l'algorithme probabiliste est le deuxième plus rapide suivi de l'algorithme dynamique. Cet ordre de résultats suit la tendance de nos résultats obtenus pour la complexité asymptomatique théorique.

Pour ce qui est de l'espace mémoire, greedy utilise un tableau supplémentaire pour stocker les blocs utilise, donc une complexité de O(m). Probabilité utilise aussi une structure supplémentaire pour stocker les blocs choisis, donc O(m). Programmation dynamique utilise un tableau qui contient dans chaque case un tuple contenant la hauteur de la tour et les blocs utiliser pour former cette tour. On a donc l'équivalent d'une structure suivant O(n^2)

Pour ce qui est de la qualité des résultats, on doit tenir en compte que l'algorithme glouton n'est pas optimal et peut rester coincé dans un maximum local et donc ne pas aboutir à la solution optimale. L'algorithme probabiliste donne en général une solution relativement proche de la solution optimale car elle peut sortir des extremum dû à la nature des probabilités, ou on peut ne pas choisir un bloc qui paraît mieux à court terme mais au final va

aboutir a une pire solution. L'algorithme de programmation dynamique par contre donne toujours la meilleure solution donc on peut se fier à la valeur qui est retourné par cet algorithme pour voir comment les autres performent.

Pour ce qui est de la difficulté d'implantation, l'algorithme glouton a été le plus simple à implémenter, suivi de l'algorithme dynamique puis l'algorithme probabiliste. L'algorithme probabiliste fut le plus difficile à implémenter, car nous avons dus employer faire appels à plusieurs opérations sur le tableau de l'exemplaire avec les librairies numpy et scipy.special, dont softmax, random et fits_on_other_block.

Selon la courbe d'erreur, combien de fois faudrait-il relancer l'algorithme probabiliste pour obtenir au moins 99 % de la hauteur optimale en moyenne?

Nous avons réalisé le test suivant:

```python
def run_samples():
    sample_size = 7500
    num_trials = 50  # Number of trials to collect data
    num_runs = 100

    optimal_height = None
    average_runs_needed = 0

    # Calculate the optimal height using dynamic programming (assuming it's the same for all runs)
    file_path = f"/Users/mazilious/git/INF8775/tp2/samples/sample_{sample_size}_1.txt"
    sys.argv = ['main.py', '-a', 'progdyn', '-e', file_path, '-p', '-t']
    optimal_height, _ = main()

    for trial in range(1, num_trials + 1):
        runs_needed = 0

        for i in range(1, num_runs + 1):
            file_name = f"/Users/mazilious/git/INF8775/tp2/samples/sample_{sample_size}_1.txt"
            file_path = os.path.join("sample", file_name)

            sys.argv = ['main.py', '-a', 'proba', '-e', file_path, '-p', '-t']
            height, execution_time = main()

            ratio = height / optimal_height
            if ratio >= 0.99:
                runs_needed = i
                break

        average_runs_needed += runs_needed
        print(f"Trial {trial} - Runs needed: {runs_needed}")

    average_runs_needed /= num_trials

    print(f"Average number of runs needed to achieve 99% of optimal height: {average_runs_needed}")

if __name__ == "__main__":
    run_samples()
```

On souhaite savoir le nombre d'exécutions moyennes de l’algorithme probabiliste afin de savoir quand est ce que la hauteur moyenne atteint 99% de la hauteur optimale. Nous avons fait un script qui roule l’algorithme de programmation dynamique pour avoir la hauteur optimale en premier puis par la suite on a exécuté l'algorithme probabiliste sur le même exemplaire en boucle. Lorsque l’on atteint 99% on garde combien de fois on a dû exécuter l’algorithme. Nous avons trouvé qu' en moyenne il nous faut 33 exécutions pour arriver a 99% de moyenne de hauteur.

Indiquez sous quelles conditions vous utiliseriez chaque algorithme.

	/ 6,5 pt

Si le temps de calcul est la condition la plus importante, l'algorithme glouton est définitivement celui à utiliser. Il faut cependant prendre le résultat avec un grain de sel, sachant qu’il est possible que le résultat fourni par cet algorithme ne soit pas le résultat parfait si est coincé sur un minimum local.

Si on veut une réponse exacte, on va choisir l'algorithme de programmation dynamique, où on obtient la hauteur optimale mais en consommant le plus de ressources. Il est aussi le plus long à développer, pas nécessairement en termes de code mais en logique.

Si on veut obtenir une réponse relativement proche à la hauteur optimale a généralement 90%, on peut choisir l’algorithme probabiliste qui est assez rapide et ne consomme pas autant de ressource et est assez facile a implémenter.`

En conclusion, si on a des restrictions temporelles et que une solution qui n’est pas nécessairement optimale, on choisit l’algorithme glouton. Lorsque l’on accepte une variabilité dans la solution avec une qualité acceptable. Bien que sa nature probabiliste puisse nécessiter un nombre d'itérations plus élevé pour converger vers une solution de haute qualité, il tend à fournir des résultats proches de l'optimal. Lorsque l’on a pas de contrainte spatio-temporelles, on choisit l'algorithme de programmation dynamique car il nous fournit la solution optimale