Don ce contre-exemple démontre que :`Next-Fit` : non optimal; `Worst-Fit` : non optimal; `Best-Fit` : optimal et `First-Fit` : optimal.

1.3 Complexité d'une solution par recherche exhaustive

Algorithm 1 Tree-Search-BF(T, k)

```
Input: Un arbre binaire de recherche T et une clé k
Output: La plus petite clé ≥ k dans l'arbre T
node ← T.root
successor ← NIL
while node ≠ NIL
    if node.key ≥ k
        successor ← node
        node ← node.left
return successor
```

Le pire cas se produit lorsque l'arbre est déséquilibré, ce qui peut se produire dans deux scénarios typiques : l'arbre dégénéré vers la droite c'est-à-dire chaque nœud n'a qu'un enfant à droite, formant une "liste" avec tous les nœuds alignés sur la droite ou l'arbre dégénéré vers la gauche c'est-à-dire chaque nœud n'a qu'un enfant à gauche, formant également une "liste" avec tous les nœuds alignés sur la gauche. La complexité dans le pire cas est donc $O(N)$. Et si l'arbre est équilibré, cette complexité devient $\mathcal{O}(\log N)$.

1.4 Le pseudo-code de la fonction `Tree-Search-FF(T, size)` et analyse de sa complexité en temps au pire cas :

Algorithm 2 Tree-Search-FF(T, size)

```
Input: Un arbre binaire de recherche T, taille du fichier size
Output: Le premier disque pouvant contenir le fichier (nœud x tel que x.space ≥ size)
if x == NIL or x.maxspace < size
        return NIL
r = TREE-SEARCH-FF(x.left, size)
if r ≠ NIL
        return r
if x.space ≥ size
        return x
return TREE-SEARCH-FF(x.right, size)
```

`Appel initial` : **Tree-Search-FF(T.root, size)**.

Même si l'algorithme optimise la recherche grâce au champ `x.maxspace`, dans le pire des cas, il devra parcourir tous les nœuds de l'arbre. Cela se produit si `x.maxspace` $\geq$ `size` pour tous les nœuds. Et si l'arbre est *dégénéré* (par exemple, tous les nœuds n'ont qu'un enfant à gauche ou à droite), il se comporte comme une liste chaînée.

Par conséquent, la complexité en temps dans le pire cas est : $\mathcal{O}(N)$, où N est le nombre total de nœuds (disques) dans l'arbre. Et si l'arbre est équilibré, cette complexité devient $\mathcal{O}(\log N)$.

1.5 Analyse la complexite en temps au pire cas des quatre algorithmes

Pour toutes les stratégies, les fichiers sont stockés dans une liste chaînée, triés en $\mathcal{O}(N \log N)$. Les disques sont stockés :

- dans une structure linéaire (une liste chaînée) pour `Next-Fit`,

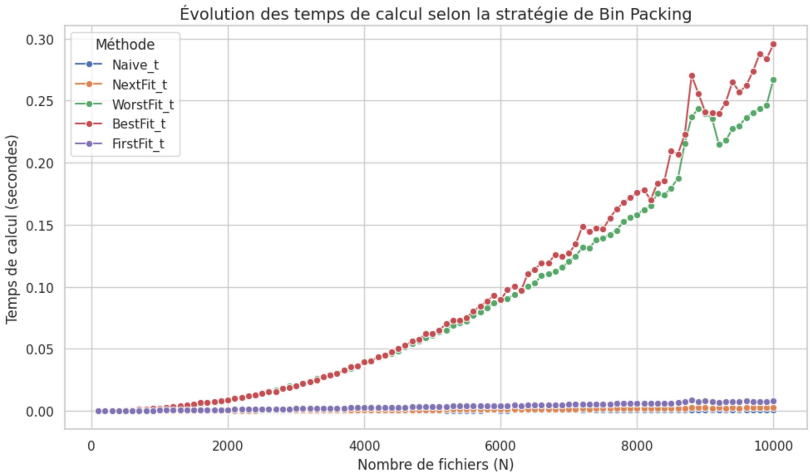

Figure 2: Évolution des temps de calcul selon la stratégie de Bin Packing

Comme attendu, les méthodes `Naive`, `Next-Fit` et `First-Fit` présentent des temps de calcul très faibles, même pour un grand nombre de fichiers. Cela est cohérent avec leur complexité algorithmique faible (souvent linéaire). En revanche, les méthodes `Best-Fit` et `Worst-Fit` nécessitent un parcours plus coûteux de la liste des disques à chaque insertion, ce qui se traduit par un temps de calcul croissant plus rapidement avec N. Cette observation est en accord avec la complexité théorique plus élevée de ces algorithmes (souvent $O(n \cdot m)$, où m est le nombre de disques ouverts et n représente le nombre de fichiers à insérer dans les disques).

Par conséquent, les résultats empiriques confirment la validité de l'analyse théorique. Les algorithmes `Next-Fit`, `First-Fit`, et Naive sont rapides (temps de calcul faible) mais peuvent être moins efficaces en termes de gaspillage. `Best-Fit` et `Worst-Fit` présentent des temps de calcul plus élevés, ce qui est attendu étant donné leur complexité plus élevée ($\mathcal{O}(N \log N)$).