Techniques algorithmiques

Ce document reprend les algorithmes vus au cours. Par convention, ils sont implémentés sous la forme de fonctions avec un nom utilisant CETTE POLICE. Si vous la voyez apparaître dans un algorithme, cela signifie donc qu'on fait appel à un autre algorithme déjà vu.

Algorithme 1 : BINPACKINGFIRSTFIT(S, K)

Entrées : une liste S de poids d'objets, une capacité K.
Sortie : une partition de S sous la forme d'une liste de listes, où la somme des tailles de chaque part n'excède pas K.

```
1  boîtes ← liste();
2  boîtes.ajouter_en_fin(liste());
3  tailles ← liste();
4  tailles.ajouter_en_fin(0);
5  n ← 0;
6  pour chaque poids ∈ S faire
       // placer l'objet dans la première boîte pouvant l'accueillir
7      trouvé ← FAUX;
8      pour chaque i allant de 0 à n faire
9          si tailles[i] + poids ≤ K alors
10             boîtes[i].ajouter_en_fin(poids);
11             tailles[i] ← tailles[i] + poids;
12             trouvé ← VRAI;
13             arrêter;
       // s'il n'y en a pas, placer l'objet dans une nouvelle boîte
14     si ¬ trouvé alors
15         boîtes.ajouter_en_fin(liste());
16         n ← n + 1;
17         boîtes[n].ajouter_en_fin(poids);
18         tailles.ajouter_en_fin(poids);
19 renvoyer boîtes;
```

Algorithme 2 : PLUSCOURTESURSÉQUENCEGLOUTONNE(S)

Entrées : un ensemble S de chaînes de caractères.
Sortie : une sur-chaîne des éléments de S.

```
1  tant que |S| > 1 faire
2      C1, C2 ← deux chaînes de S de chevauchement maximal;
3      S ← S \ {C1, C2}∪ plus courte sur-chaîne de C1 et C2;
4  renvoyer l'unique élément de S;
```

Algorithme 3 : FibonacciNaïf(n)

Entrées : un naturel n.
Sortie : le n-ème nombre de Fibonacci.

```
1 si n ≤ 1 alors renvoyer n;
2 renvoyer FibonacciNaïf(n − 1) + FibonacciNaïf(n − 2);
```

Algorithme 4 : FibonacciStockage(n)

Entrées : un naturel n.
Sortie : le n-ème nombre de Fibonacci.

```
1 si n ≤ 1 alors renvoyer n;
2 valeurs ← tableau(n + 1, 0);
3 valeurs[0] ← 0; valeurs[1] ← 1;
4 pour chaque i allant de 2 à n faire
5 |  valeurs[i] ← valeurs[i − 1] + valeurs[i − 2];
6 renvoyer valeurs[n];
```

Algorithme 5 : DécoupeOptimaleNaïve(longueur, prix)

Entrées : une longueur naturelle de tige, un tableau de prix donnant pour chaque longueur possible le prix correspondant.
Sortie : le profit maximal réalisable.

```
/* aucune découpe possible: renvoyer directement le prix                    */
1 si longueur ≤ 1 alors renvoyer prix[longueur];
/* tester toutes les découpes possibles et garder la meilleure              */
2 meilleur_prix ← prix[longueur];
3 pour chaque i allant de 1 à longueur −1 faire
4 |  meilleur_prix ← max(meilleur_prix, prix[i] + DécoupeOptimaleNaïve(longueur −i , prix));
5 renvoyer meilleur_prix;
```

Algorithme 6 : DécoupeOptimaleProgDyn(longueur, prix)

Entrées : une longueur naturelle de tige, un tableau de prix donnant pour chaque longueur possible le prix correspondant.
Sortie : le profit maximum réalisable.

```
1 profits ← prix;
2 pour chaque k allant de 2 à longueur faire
3 |  pour chaque i allant de 1 à k − 1 faire
4 |  |  profits[k] ← max(profits[k], prix[i] + profits[k − i]);
5 renvoyer profits[longueur];
```

Algorithme 7 : DistanceEditionNaïve(S, k, T, n)

Entrées : deux chaînes de caractères S et T et leurs longueurs (respectivement k et n)
Sortie : la distance d'édition entre S et T

```
1 si k = 0 alors renvoyer n;
2 si n = 0 alors renvoyer k;
3 option1 ← 1 + DistanceEditionNaïve(S, k − 1, T, n);
4 option2 ← 1 + DistanceEditionNaïve(S, k, T, n − 1);
5 option3 ← (S[k − 1] ≠ T[n − 1]) + DistanceEditionNaïve(S, k − 1, T, n − 1);
6 renvoyer min(option1, option2, option3);
```

Algorithme 8 : DistanceEditionProgDyn(S, T)

Entrées : deux chaînes de caractères S et T et leurs longueurs (respectivement k et n)
Sortie : la distance d'édition entre S et T

```
/* initialiser la matrice de coûts avec les cas de base                    */
k ← |S|;
n ← |T|;
coûts ← matrice de zéros de dimension (k + 1) × (n + 1);
pour chaque i allant de 0 à k faire coûts[i][0] ← i ;
pour chaque j allant de 0 à n faire coûts[0][j] ← j ;
/* remplir la matrice de coûts à l'aide de la récurrence                   */
pour chaque i allant de 1 à k faire
    pour chaque j allant de 1 à n faire
        option1 ← 1 + coûts[i − 1][j];
        option2 ← 1 + coûts[i][j − 1];
        option3 ← (S[i − 1] ≠ T[j − 1]) + coûts[i − 1][j − 1];
        coûts[i][j] ← min(option1, option2, option3);
renvoyer coûts[k][n];
```

Algorithme 9 : DécoupeOptimaleProgDynSol(longueur, prix)

Entrées : une longueur naturelle de tige, un tableau de prix donnant pour chaque longueur possible le prix correspondant.
Sortie : les profits maximaux réalisables pour chaque longueur de tige, ainsi que la longueur du dernier morceau pour chaque découpe optimale.

```
profits ← prix;
taille_dernière_pièce ← tableau d'entiers de longueur |prix|;
pour chaque k allant de 2 à longueur faire
    bénéfice ← prix[k];
    taille_dernière_pièce[k] ← k;
    pour chaque i allant de 1 à k − 1 faire
        si prix[i] + profits[k − i] > bénéfice alors
            bénéfice ← prix[i] + profits[k − i];
            taille_dernière_pièce[k] ← i;
    profits[k] ← bénéfice;
renvoyer (profits, taille_dernière_pièce);
```

Algorithme 10 : MeilleurParent(matrice, i, j, coût_différence)

Entrées : une matrice de coûts, un numéro i de ligne, un numéro j de colonne, et le coût d'une différence entre deux caractères.

Sortie : les coordonnées du parent de matrice$[i][j]$ de coût minimum.

```
   // enregistrer les transitions possibles et leur coût
 1 transitions ← tableau_associatif();
 2 si i ≥ 1 et j ≥ 1 alors transitions[(i − 1, j − 1)] ← matrice[i − 1][j − 1] + coût_différence ;
 3 si i ≥ 1 et j ≥ 0 alors transitions[(i − 1, j)] ← matrice[i − 1][j] + 1;
 4 si i ≥ 0 et j ≥ 1 alors transitions[(i, j − 1)] ← matrice[i][j − 1] + 1 ;
   // sélectionner le minimum et renvoyer ses coordonnées
 5 coordonnées ← NIL;
 6 minimum_actuel ← +∞;
 7 pour chaque coords, valeur ∈ transitions faire
 8 |   si valeur < minimum_actuel alors
 9 |   |   minimum_actuel ← valeur;
10 |   |   coordonnées ← coords;
11 renvoyer coordonnées;
```