Ministre de l'Enseignement Supérieur et de la Recherche
Université Sorbonne Paris Nord - Villetaneuse -
Institut Ingénieur Sup Galilée

Devoir N°01

Module Algorithmique

Année universitaire 2024/2025

Date du devoir: 08 Décembre 2024

Nom: SEBA
Prénom: Mohammed Rabie
Identifiant: 12409891

Contents

1 Introduction

Ce document constitue le rapport pour le devoir d'algorithmique concernant le problème de la chasse au trésor. Nous analyserons les différentes questions posées, fournirons les algorithmes en pseudo-code ainsi qu'une implémentation en langage C, et nous effectuerons des tests sur des données données.

2 Analyse du problème

Le problème consiste à trouver le chemin qui maximise la valeur des objets collectés dans un bâtiment représenté sous forme de grille $n \times m$. Le chasseur commence en $(1, 1)$ et doit atteindre (n, m) en se déplaçant uniquement vers la droite ou vers le bas.

2.1 Borne sur le nombre de solutions admissibles

1. Nombre de chemins entre (i, i) et $(i + 1, i + 1)$

Il existe deux chemins possibles :

- Aller à droite, puis vers le bas.
- Aller vers le bas, puis à droite.

Donc, le nombre de chemins est 2.

2. Nombre de chemins entre (i, i) et $(i + 2, i + 2)$ passant par $(i + 1, i + 1)$

Chaque segment $(i, i) \to (i + 1, i + 1)$ a 2 choix de chemin (comme montré ci-dessus), et de même pour $(i + 1, i + 1) \to$(i+2, i+2). Par conséquent, le nombre total est $2 \times 2 = 4$ chemins.

3. Nombre de chemins entre $(1, 1)$ et (n, n) passant par toutes les cases (i, i)

Pour traverser toutes les cases de la diagonale (i, i), il n'y a qu'une seule possibilité de chemin à chaque étape : avancer d'une étape diagonale. Ainsi, il n'y a que ce chemin là.

4. Nombre de chemins entre $(1, 1)$ et (n, m)

Le problème revient à calculer les chemins dans une grille de taille $(n - 1) \times (m - 1)$. Ce nombre est donné par le coefficient binomial :

$$\binom{n + m - 2}{n - 1} = \frac{(n + m - 2)!}{(n - 1)!(m - 1)!}$$

5. Peut-on envisager une approche par énumération ?

Non, l'approche par énumération est irréaliste lorsque n et m sont grands, car le nombre de chemins possibles croît exponentiellement. Une telle approche serait inefficace en pratique.

3 Approche récursive (Section 2.2)

3.1 Formulation récursive

- Si le dernier déplacement est vers le bas :

$$\text{tresor}(i, j) = \text{valeur}[i, j] + \text{tresor}(i + 1, j)$$

- Si le dernier déplacement est vers la droite :

$$\text{tresor}(i, j) = \text{valeur}[i, j] + \text{tresor}(i, j + 1)$$

Dans le cas général, on a :

$$\text{tresor}(i, j) = \text{valeur}[i, j] + \max(\text{tresor}(i + 1, j), \text{tresor}(i, j + 1))$$

3.2 Cas de base

- $\text{tresor}(n, m) = \text{valeur}[n, m]$
- $\text{tresor}(i, m) = \text{valeur}[i, m] + \text{tresor}(i + 1, m)$ (bord droit)
- $\text{tresor}(n, j) = \text{valeur}[n, j] + \text{tresor}(n, j + 1)$ (bord inférieur)

3.3 Complexité

L'approche récursive a une complexité exponentielle, car elle effectue de nombreux appels redondants en recalculant les sous-problèmes déjà résolus.

3.3.1 Analyse de la complexité de l'algorithme récursif

Pour montrer que la complexité de l'algorithme récursif est au moins exponentielle, considérons le cas particulier où $n = m$, c'est-à-dire une grille carrée de taille $n \times n$.

3.3.2 Formulation de la fonction récursive

L'algorithme récursif pour calculer $\text{tresor}(i, j)$ est donné par :

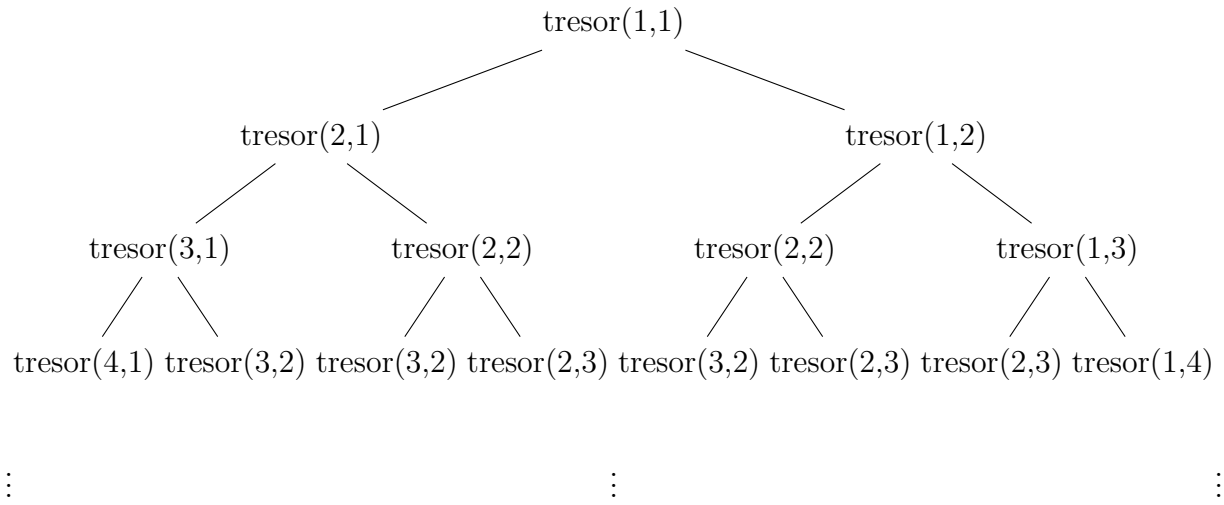
$$\text{tresor}(i, j) = \text{valeur}[i, j] + \max(\text{tresor}(i + 1, j), \text{tresor}(i, j + 1))$$

La fonction effectue deux appels récursifs à chaque étape, sauf dans les cas suivants

Voir en haut la partie Cas de base 3.2 !

3.3.3 Arbre des appels récursifs

Chaque appel récursif génère deux sous-problèmes, formant ainsi un arbre binaire. La profondeur de cet arbre est donnée par $2n - 1$ (longueur de la diagonale complète).



⋮ ⋮ ⋮

3.3.4 Nombre total d'appels

Pour une grille $n \times n$, le nombre total d'appels récursifs est proportionnel au nombre total de chemins possibles entre $(1, 1)$ et (n, n). Ce nombre est donné par :

$$\binom{2n-2}{n-1} = \frac{(2n-2)!}{(n-1)!(n-1)!}$$

Ce résultat montre que le nombre total d'appels récursifs croît de manière exponentielle avec n.

3.3.5 Nombre de comparaisons (`max`)

À chaque nœud de l'arbre, une comparaison max est effectuée. Le nombre total de comparaisons est directement proportionnel au nombre d'appels récursifs, soit $\binom{2n-2}{n-1}$.

3.3.6 Conclusion

L'arbre des appels récursifs et le calcul du nombre de comparaisons montrent que la complexité de l'algorithme récursif est **exponentielle**.

3.4 Algorithme récursif en pseudo-code

```
function tresor(i, j):
    if i == n and j == m:
        return valeur[n][m]
    if i == n:
        return valeur[i][j] + tresor(i, j+1)
    if j == m:
        return valeur[i][j] + tresor(i+1, j)
    return valeur[i][j] + max(tresor(i+1, j), tresor(i, j+1))
```

4 Programmation dynamique (Section 2.3)

4.1 Algorithme dynamique utilisant le tableau *gain[i][j]*

Nous modifions l'algorithme récursif pour utiliser un tableau *gain*, initialisé à -1, afin de mémoriser les résultats des sous-problèmes et d'éviter les calculs redondants.

```
function tresor(i, j):
    if gain[i][j] != -1:
        return gain[i][j]
    if i == n and j == m:
        gain[i][j] = valeur[n][m]
    elif i == n:
        gain[i][j] = valeur[i][j] + tresor(i, j+1)
    elif j == m:
        gain[i][j] = valeur[i][j] + tresor(i+1, j)
    else:
        gain[i][j] = valeur[i][j] + max(tresor(i+1, j), tresor(i,
            j+1))
    return gain[i][j]
```

4.2 Ordre de calcul des cases de *gain*

Pour garantir que les résultats nécessaires soient toujours disponibles, les cases de *gain* doivent être calculées en partant de la fin (coin inférieur droit de la grille) et en remontant vers le début (coin supérieur gauche).

4.3 Algorithme itératif basé sur la programmation dynamique

```
function calculerGain(valeur, n, m):
    initialize gain with -1
    for i = n downto 1:
        for j = m downto 1:
            if i == n and j == m:
                gain[i][j] = valeur[i][j]
            elif i == n:
                gain[i][j] = valeur[i][j] + gain[i][j+1]
            elif j == m:
                gain[i][j] = valeur[i][j] + gain[i+1][j]
            else:
                gain[i][j] = valeur[i][j] + max(gain[i+1][j],
                   gain[i][j+1])
    return gain[1][1]
```

4.4 Complexité de l'algorithme

La complexité de cet algorithme est $O(n \times m)$, car chaque case est calculée une seule fois.

4.5 Chemin optimal

Pour stocker le chemin optimal, nous pouvons maintenir un tableau *chemin* qui enregistre la direction prise à chaque étape (droite ou bas).

```
function calculerGainAvecChemin(valeur, n, m):
    initialize gain with -1
    initialize chemin with empty values
    for i = n downto 1:
        for j = m downto 1:
            if i == n and j == m:
                gain[i][j] = valeur[i][j]
            elif i == n:
                gain[i][j] = valeur[i][j] + gain[i][j+1]
                chemin[i][j] = 'D'
            elif j == m:
                gain[i][j] = valeur[i][j] + gain[i+1][j]
                chemin[i][j] = 'B'
            else:
                if gain[i+1][j] > gain[i][j+1]:
                    gain[i][j] = valeur[i][j] + gain[i+1][j]
                    chemin[i][j] = 'B'
                else:
                    gain[i][j] = valeur[i][j] + gain[i][j+1]
                    chemin[i][j] = 'D'
    return gain[1][1], chemin
```

5 Variante du problème (Section 2.4)

5.1 Condition sur la valeur de k

Pour que le chasseur puisse sortir du bâtiment en visitant au plus k salles, il doit exister un chemin reliant $(1,1)$ à (n,m) comportant un nombre de salles $\leq k$. Cela impose la condition :

$$k \geq n + m - 1$$

car $n + m - 1$ est le nombre minimum de salles visitées dans un chemin direct.

5.2 Valeurs de k pour une solution optimale évidente

- Si $k = n + m - 1$, le chasseur doit obligatoirement suivre un chemin direct sans détour.
- Si $k > n + m - 1$, il est possible d'explorer des chemins alternatifs pour maximiser la valeur collectée.

5.3 Adaptation des algorithmes

Pour prendre en compte les nouveaux déplacements (haut et gauche) et la contrainte de k salles, les algorithmes doivent être modifiés :

5.3.1 Adaptation de l'algorithme récursif

```
function tresor(i, j, k):
    if k < 0:
        return -infini
    if i == n and j == m:
        return valeur[n][m]
    gain_bas = tresor(i+1, j, k-1) if i+1 <= n else -ifini
    gain_droite = tresor(i, j+1, k-1) if j+1 <= m else -infini
    gain_haut = tresor(i-1, j, k-1) if i-1 >= 1 else -infini
    gain_gauche = tresor(i, j-1, k-1) if j-1 >= 1 else -infini
    return valeur[i][j] + max(gain_bas, gain_droite, gain_haut,
       gain_gauche)
```

5.3.2 Adaptation de l'algorithme itératif avec programmation dynamique

```
function calculerGain(valeur, n, m, k):
    for i = 1 to n:
        for j = 1 to m:
            for p = 0 to k:
                gain[i][j][p] = -infini
    gain[n][m][0] = valeur[n][m]
    for p = 1 to k:
        for i = n downto 1:
            for j = m downto 1:
                gain_bas = gain[i+1][j][p-1] if i+1 <= n else -
                   infini
                gain_droite = gain[i][j+1][p-1] if j+1 <= m else
                   -infini
                gain_haut = gain[i-1][j][p-1] if i-1 >= 1 else -
                   infini
                gain_gauche = gain[i][j-1][p-1] if j-1 >= 1 else
                   -infini
                gain[i][j][p] = valeur[i][j] + max(gain_bas,
                   gain_droite, gain_haut, gain_gauche)
    return max(gain[1][1][p] for p in range(k+1))
```

6 Utilisation des fichiers et exécution

Trois fichiers C ont été fournis correspondant aux différentes implémentations des algorithmes développés dans ce rapport. Voici leur description et les étapes pour les exécuter:

6.1 Fichier `recursif.c`

Ce fichier contient l'implémentation de l'algorithme récursif (section 2.2). Cet algorithme calcule la valeur maximale des objets collectés en explorant toutes les possibilités, mais il est inefficace pour les grandes grilles en raison de sa complexité exponentielle.

Commande pour compiler :

```
gcc recursif.c -o recursif
```

Commande pour exécuter :

```
./recursif
```

—

6.2 Fichier `iteratif.c`

Ce fichier contient l'implémentation de l'algorithme itératif basé sur la programmation dynamique (section 2.3). Cet algorithme est optimisé pour résoudre le problème en temps $O(n \times m)$ et affiche également le chemin optimal parcouru.

Commande pour compiler :

```
gcc iteratif.c -o iteratif
```

Commande pour exécuter :

```
./iteratif
```

—

6.3 Fichier `avec_k.c`

Ce fichier contient l'algorithme développé pour la variante bonus (section 2.4), qui inclut des déplacements supplémentaires (haut et gauche) et une contrainte sur le nombre maximum de salles visitées (`k`). L'utilisateur peut modifier la valeur de `k` directement dans le fichier.

Commande pour compiler :

```
gcc avec_k.c -o avec_k
```

Commande pour exécuter :

```
./avec_k
```

—

6.4 Structure du fichier compressé

Les fichiers suivants sont inclus dans l'archive `.zip` :

- `recursif.c` : Implémentation récursive.
- `iteratif.c` : Implémentation itérative avec programmation dynamique.
- `avec_k.c` : Implémentation pour la variante bonus.
- `rapport.pdf` : Ce document expliquant le projet.