Exercice 3 :

1) a) Nous avons écrit le pseudo-code suivant :

Dans un tableau contenant des 0 et des 1, on recherche le plus grand sous-tableau qui contient autant de 0 que de 1. Ce sous-tableau doit donc avoir une moyenne de 0.5, donc un total égal à la moitié de sa taille.

On teste donc pour chaque début de sous-tableau possible, que l'on note i, le plus grand sous-tableau qu'on peut atteindre, donc la plus grande valeur de fin que l'on peut atteindre, que l'on note j. On enregistre chaque j obtenu dans un tableau, et on prend le maximum de ce tableau.

Pour éviter de recalculer la somme de chaque sous-tableau que l'on teste, on enregistre la dernière somme, et on ajoute la dernière valeur, puisque l'on agrandit à chaque tour le sous-tableau d'une case. De plus, on ne teste pas le sous-tableau lorsqu'il est de taille impair, puisqu'il n'y a alors pas autant de 0 que de 1. On teste donc que quand $j - i + 1$ est pair.

```
int tailles[l] = {0}

// Première partie

Pour chaque i appartenant à [0, l-1[ {

    t = T[i]

    Pour chaque j appartenant à [i, l[ {
        t += T[i]

        Si (j-i+1) est pair{
            tailles[i] = j - i + 1
        }
    }
}

i, j = indice_max, max (tailles)

retourner T[i:j]
```

b) Dans la première partie, il y a l^2 itérations de complexités respectives $O(1)$, la complexité de cette partie est donc $O(l^2)$. Dans le seconde partie, on détermine le (premier) maximum du tableau et son indice, on suppose que l'algorithme de recherche est alors de complexité linéaire. Finalement la complexité de l'algorithme est $O(l^2)$.

2) a) On réalise l'algorithme suivant :

On souhaite effectuer le pré-calcul suivant :

1 - On mesure le déséquilibre de chaque case (en commençant à 0)

2 - On sauvegarde dans un tableau D dont les indices "vont de -l à l" et donc de taille 2*l+1, le premier indice pour lequel chaque déséquilibre est atteint. Les valeurs -1 signifient alors que la valeur n'est jamais atteinte

Par exemple :

T : [0, 1, 0, 0, 1]

-> C : $[-1, 0, -1, -2, -1]$

-> D : [3, 0, 1, −1, −1]

La solution est alors, de d'abord parcourir le tableau T pour déterminer le déséquilibre de chaque case, en utilisant le déséquilibre de la dernière case pour éviter de reparcourir le tableau. Ensuite, on crée un tableau D rempli initialement de −1, on parcourt le tableau C, et pour chaque valeur, si D[i]=−1 on la remplace, sinon on continue.

```
int C[l];

int t = 0

Pour i appartenant à [0, l[ {
    Si T[i] == 1 {
        t+=1
    } Sinon {
        t-=1
    }

    C[i] = t
}

int D[2*l+1] = {-1};

Pour i appartenant à [0, l[ {
    Si D[C[i]] == -1 {
        D[C[i]] = i
    }
}

retourner D
```

b) Cet algorithme exécute une première boucle de l itérations, de complexités respectives O(1), elle a donc une complexité O(l). Il exécute ensuite une seconde boucle de l itérations, de complexités respectives O(1), elle a donc aussi une complexité O(l). Enfin, le reste de l'algorithme a une complexité O(1).

Il a donc une complexité finale de O(l).

3) Une solution pourrait être de parcourir la table de pré-calcul, de gauche à droite et de droite à gauche parallèlement, en notant les premiers indices rencontrés, de valeurs opposées.