Bài 3

a)

Cho danh sách `l` gồm điểm thi hai môn `X, Y` của `n` sinh viên (SV), trong đó `SV_i` thứ `i` với `1 <= i <= n` sẽ có một cặp điểm `(x_i, y_i)` là các số thực thuộc `[0;10]` (để đơn giản, ta xét tất cả giá trị đều phân biệt). Yêu cầu đặt ra là cần đếm số cặp `(SV_r, SV_j)` mà `SV_j` có điểm trội hơn so với `SV_r`, tức là `x_j > x_i` và `y_j > y_i`. Ta muốn xây dựng thuật toán divide-and-conquer để giải quyết bài toán này với ý tưởng sơ lược như sau:

- **Bước divide**: sắp xếp `l` theo điểm môn `X` rồi chia thành hai nửa bằng nhau là `l_left`, `l_right`.
- **Bước conquer**: tiếp theo, đếm số cặp trội hơn giữa `l_left`, `l_right` bằng cách sắp xếp mỗi danh sách con theo môn `Y`. Ta xét hai `SV` có điểm môn `Y` thấp nhất ở cả hai danh sách rồi thực hiện so sánh và đếm, mỗi bước sẽ loại đi được một trong hai `SV` đó, cứ tiếp tục cho đến hết.

Hãy mô tả cụ thể hơn các ý tưởng ở trên để xây dựng chi tiết thuật toán divide-and-conquer giải quyết bài này. Từ đó, thiết lập công thức đệ quy để ước lượng số phép so sánh. Từ đó sử dụng định lý Master, ước lượng độ phức tạp của thuật toán dựa theo hệ thức tìm được.

In [ ]:
```python
def crossCountPair(arr, l, mid, r):
    arr[l:mid+1] = sorted(arr[l:mid+1], key=lambda p: p[1])
    arr[mid+1:r+1] = sorted(arr[mid+1:r+1], key=lambda p: p[1])

    i, j = l, mid+1
    count = 0
    while i <= mid and j <= r:
        if arr[i][1] < arr[j][1]:
            count += r - j + 1
            i += 1
        else:
            j += 1
    return count

def countPair(arr, l, r):
    if r - l + 1 <= 1:
        return 0
    mid = (r + l) // 2
    leftCount = countPair(arr, l, mid)
    rightCount = countPair(arr, mid + 1, r)
    crossCount = crossCountPair(arr, l, mid, r)
    return leftCount + rightCount + crossCount

students = [(2, 3), (1, 5), (4, 2), (3, 6), (5, 4)]
students = sorted(students, key=lambda s: s[0])
countPair(students, 0, len(students)-1)
```

Out[ ]:
```
4
```

Giải bài toán sử dụng Định lý Master

- Mảng gốc được chia thành 2 mảng con kích thước $\frac{n}{2}$. Mỗi mảng con này được xử lý đệ quy, dẫn đến $2T\left(\frac{n}{2}\right)$.
- Sau khi chia, cần sắp xếp hai mảng con này. Việc sử dụng Merge Sort để sắp xếp tốn $O(n \log n)$.
- Sau đó, hàm `crossCountPair` duyệt và so sánh các phần tử để đếm số cặp trội hơn, tốn $O(n)$.

Vì vậy, công thức tổng quát cho độ phức tạp thời gian là:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$

Ở đây:

- $a = 2$
- $b = 2$
- $f(n) = O(n \log n)$

$n \log n$ có thể được viết lại dưới dạng: $n \log n = n^1 \cdot \log n$

Do đó, ta có: $n \log n \approx n^1 \cdot n^\epsilon = n^{1+\epsilon}$

Ở đây, ϵ là một giá trị rất nhỏ và $n^{1+\epsilon}$ là một xấp xỉ hợp lý cho $n \log n$ khi n lớn.

Vì $d = 1 + \epsilon$, trong đó ϵ là một giá trị nhỏ nhưng dương, ta có:

$$b^d = 2^{1+\epsilon} > 2 = a$$

Kết luận: Theo Định lý Master, độ phức tạp thời gian của thuật toán được xác định là $T(n) = O(n \log n)$

b)

Cho mã giả bên dưới mô tả một thuật toán chia để trị `process` thao tác trên một mảng số thực `A[0..n-1]` (được khai báo toàn cục) như sau:

```
pseudo
procedure process(L, R):
    if L == R return
    mid <- (L+R)/2
    mid1 <- (L+mid)/2, mid2 <- (mid+R)/2
    process(L, mid)
    process(mid, R)
    process(mid1, mid2)
    for i = mid1 to mid2:
        A[i] <- A[i] * mid1 - i * mid2
    sort array A from index mid1 to index mid2
```

Xây dựng công thức đệ quy cho `T(n)` mô tả chi phí tính toán khi gọi `process(0,n-1)` và đánh giá độ phức tạp cho thuật toán trên trong hai trường hợp sau: thuật toán sort được sử dụng là Selection sort / thuật toán sort được sử dụng là Merge sort.

1. Công thức đệ quy

$$T(n) = 3T\left(\frac{n}{2}\right) + g(n)$$

Với:

- $a = 3, b = 2$

2. Trường hợp sử dụng Merge Sort:

- $g(n) = n \log n + \frac{n}{2}$

Ta có:

$$\lim_{n\to\infty} \frac{a * g\left(\frac{n}{b}\right)}{g(n)} = \lim_{n\to\infty} \frac{\frac{3n}{2}\log\frac{n}{2} + \frac{3n}{4}}{n \log n + \frac{n}{2}} = \frac{3}{2} > 1$$

Kết luận:

$$T(n) \in \mathcal{O}\left(n^{\log_2 3}\right)$$

3. Trường hợp sử dụng Selection Sort:

- $g(n) = n^2 + \frac{n}{2}$

Ta có:

$$\lim_{n\to\infty} \frac{g\left(\frac{n}{b}\right)}{g(n)} = \lim_{n\to\infty} \frac{\frac{3n^2}{4} + \frac{3n}{4}}{n^2 + \frac{n}{2}} = \frac{3}{4} < 1$$

Kết luận:

$$T(n) \in \mathcal{O}(n^2)$$

c)

Thuật toán Strassen

Thuật toán Strassen là một thuật toán chia để trị, dùng để tìm tích C của hai ma trận vuông A, B có cùng kích thước $n \times n$ được mô tả như bên dưới: đầu tiên "cắt" ma trận A, B thành 4 khối kích thước $\frac{n}{2} \times \frac{n}{2}$ (nếu n lẻ thì ta sẽ thêm 1 dòng/cột gồm toàn số 0 vào cuối để chia được) như sau:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Thuật toán Strassen sẽ tính:

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22}) \times B_{11} \\ M_3 &= A_{11} \times (B_{12} - B_{22}) \\ M_4 &= A_{22} \times (B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12}) \times B_{22} \\ M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \end{aligned}$$

Từ đó sẽ tính được ma trận C theo công thức:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

Yêu cầu: Giả sử các thao tác cộng, trừ hai ma trận cùng kích thước $m \times m$ sẽ có chi phí là $\Theta(m^2)$ với mọi $m \in \mathbb{Z}^+$, mô tả công thức đệ quy cho $T(n)$ thích hợp để xác định chi phí tính toán ở trên. Từ đó, dùng định lý Master để ước lượng độ phức tạp cho $T(n)$ và so sánh sự hiệu quả của thuật toán này với thuật toán naive thông thường để nhân hai ma trận $n \times n$.

1. Công thức đệ quy cho $T(n)$

Gọi $T(n)$ là chi phí tính toán để nhân hai ma trận kích thước $n \times n$.

Với mỗi bước chia để trị:

- Mỗi lần chia, ma trận được chia thành 4 phần kích thước $\frac{n}{2} \times \frac{n}{2}$.
- Có 7 phép nhân ma trận con kích thước $\frac{n}{2} \times \frac{n}{2}$ cần thực hiện.
- Các phép cộng và trừ ma trận có độ phức tạp là $O(n^2)$, cụ thể là $f(n) = 18n^2$ cho các phép cộng và trừ ma trận.

Do đó, công thức đệ quy cho $T(n)$ là:

$$T(n) = 7T\left(\frac{n}{2}\right) + 18n^2$$

2. Ước lượng độ phức tạp của $T(n)$ bằng định lý Master

Định lý Master được áp dụng cho các công thức đệ quy dạng:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

Trong đó:

- $a = 7$
- $b = 2$
- $f(n) = 18n^2 \in O(n^2) \Rightarrow d = 2$

Như vậy: $a = 7 > 2^2 = b^d$

Theo Định lý Master: $T(n) = O(n^{\log_2 7}) \approx O(n^{2.807})$

3. So sánh với thuật toán nhân ma trận thông thường

- Thuật toán nhân ma trận thông thường có độ phức tạp là $O(n^3)$.
- Thuật toán Strassen có độ phức tạp $O(n^{2.807})$, tốt hơn so với $O(n^3)$, đặc biệt là khi kích thước ma trận n lớn.

Bài 4:

a)

Xét bài toán sau đây: cho số nguyên $n \geq 2$ và mảng $a_0, a_1, \ldots, a_{n-1}$ gồm các số nguyên có giá trị tuyệt đối không quá 10^9. Tìm giá trị lớn nhất của hiệu $a_j - a_i$ với $0 \leq i < j < n$.

Ví dụ: nếu có mảng $a[6] = \{5, 1, -3, 0, 3, -4\}$ thì đáp số sẽ là 6 (lấy $a_j - a_i = 3 - (-3) = 6$).

- Hãy mô tả ngắn gọn cách giải vét cạn brute-force cho bài toán trên (dùng mã giả hoặc code C++/Python) và đánh giá độ phức tạp tương ứng của thuật toán đó.

- Cho ý tưởng về thuật toán "chia để trị" để giải bài toán trên như sau: chia mảng ban đầu thành 2 mảng con từ trung vị mid để tính giá trị lớn nhất của $a_j - a_i$ trong 3 trường hợp:

  1. $0 \leq i < j < \text{mid}$
  2. $\text{mid} \leq i < j < n$
  3. $0 \leq i < \text{mid} \leq j < n$

Đối với trường hợp (1) và (2), ta thực hiện gọi đệ quy giải hai bài toán riêng biệt cho hai mảng con tương ứng. Còn với trường hợp (3), ta chỉ cần lấy giá trị lớn nhất trong các số từ $\text{mid} \to n$ (đặt là $\max_R$) rồi trừ cho giá trị nhỏ nhất trong các số từ $0 \to \text{mid}$ (đặt là $\min_L$) là được; chỉ cần phải duyệt để tìm giá trị lớn nhất ($\max_R$) và nhỏ nhất ($\min_L$) trong các mảng con qua một vòng lặp. Yêu cầu: mô tả công thức đệ quy cho $T(n)$ thích hợp để xác định chi phí tính toán trong thuật toán nêu trên. Từ đó, sử dụng định lý Master để đánh giá độ phức tạp của $T(n)$.

- Biết rằng có thể sử dụng cấu trúc dữ liệu "segment tree" (cây phân đoạn) để giải bài toán này (range minimum/maximum queries) (truy vấn min/max trên nhiều mảng con), trong đó độ phức tạp là $O(\log n)$ cho việc duyệt cây một lần duy nhất và chi phí $O(\log n)$ để tìm giá trị lớn nhất và nhỏ nhất cho các phần tử trên mỗi mảng con. Hỏi nếu áp dụng cấu trúc dữ liệu này như thế thì cải thiện độ phức tạp của thuật toán nêu trên như thế nào (mô tả lại công thức cho $T(n)$ và đánh giá)?

In [ ]:
```python
def max_difference_brute_force(arr):
    maxVal = -1
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if arr[j] - arr[i] > maxVal:
                maxVal = abs(arr[j] - arr[i])
    return maxVal
arr = [5, 1, -3, 0, 3, -4]
print("Max difference (Brute-force):", max_difference_brute_force(arr))
```

```
Max difference (Brute-force): 6
```

Brute-force:

- Duyệt qua tất cả các cặp i, j với $0 \leq i < j < n$ và tính hiệu $a_j - a_i$.
- Độ phức tạp của thuật toán là $O(n^2)$ vì cần hai vòng lặp lồng nhau để kiểm tra tất cả các cặp phần tử.

In [ ]:
```python
def max_difference_divide_and_conquer(arr, left, right):
    if left == right:
        return 0
    mid = (right + left) // 2

    maxDiffLeft = max_difference_divide_and_conquer(arr, left, mid)
    maxDiffRight = max_difference_divide_and_conquer(arr, mid + 1, right)
    minLeft = min(arr[left:mid+1])
    maxRight = max(arr[mid+1:right+1])

    return max(maxRight - minLeft, max(maxDiffLeft, maxDiffRight))
```

```
arr = [5, 1, -3, 0, 3, -4]
print("Max difference (Divide and Conquer):", max_difference_divide_and_conquer(arr, 0, len(arr)-1))
```

```
Max difference (Divide and Conquer): 6
```

Chia để trị:

- Ý tưởng là chia mảng thành hai phần và giải quyết bài toán bằng cách tính hiệu lớn nhất ở mỗi nửa mảng và hiệu lớn nhất giữa phần tử lớn nhất bên phải và phần tử nhỏ nhất bên trái.
- Cụ thể, công thức đệ quy là:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

- Như vậy: $a = 2, b = 2, d = 1$
- Độ phức tạp của thuật toán là $O(n \log n)$ vì $a = 2 = 2^1 = b^d$

In [ ]:
```
class SegmentTree:
    def __init__(self, data):
        self.n = len(data)
        self.tree_min = [0] * (2 * self.n)
        self.tree_max = [0] * (2 * self.n)
        self.build(data)

    def build(self, data):
        for i in range(self.n):
            self.tree_min[self.n + i] = data[i]
            self.tree_max[self.n + i] = data[i]
        for i in range(self.n - 1, 0, -1):
            self.tree_min[i] = min(self.tree_min[2 * i], self.tree_min[2 * i + 1])
            self.tree_max[i] = max(self.tree_max[2 * i], self.tree_max[2 * i + 1])

    def range_min(self, l, r):
        l += self.n
        r += self.n
        min_val = float('inf')
        while l < r:
            if l & 1:
                min_val = min(min_val, self.tree_min[l])
                l += 1
            if r & 1:
                r -= 1
                min_val = min(min_val, self.tree_min[r])
            l //= 2
            r //= 2
        return min_val

    def range_max(self, l, r):
        l += self.n
        r += self.n
        max_val = float('-inf')
        while l < r:
            if l & 1:
                max_val = max(max_val, self.tree_max[l])
                l += 1
            if r & 1:
                r -= 1
                max_val = max(max_val, self.tree_max[r])
            l //= 2
            r //= 2
        return max_val

def max_difference_segment_tree(arr, segment_tree, left, right):
    if left == right:
        return 0
    mid = (right + left) // 2

    maxDiffLeft = max_difference_divide_and_conquer(arr, left, mid)
    maxDiffRight = max_difference_divide_and_conquer(arr, mid + 1, right)
    minLeft = segment_tree.range_min(left, mid+1)
    maxRight = segment_tree.range_max(mid+1, right+1)


    return max(maxRight - minLeft, max(maxDiffLeft, maxDiffRight))
```

```
# Ví dụ sử dụng:
arr = [5, 1, -3, 0, 3, -4]
segment_tree = SegmentTree(arr)
print("Max difference (Segment Tree):", max_difference_segment_tree(arr, segment_tree, 0, len(arr)-1))
```

```
Max difference (Segment Tree): 6
```

- Khi sử dụng Segment Tree, việc tìm giá trị lớn nhất và nhỏ nhất trong mảng từ 0 đến mid và từ mid đến n sẽ tốn $O(\log n)$.
- Do đó, công thức đệ quy có dạng:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(\log n)$$

- Ở đây, $a = 2$, $b = 2$, và $f(n) = O(\log n)$.
- Ta có:
  - $f(n) = O(\log n)$, tức là $f(n) = O(n^0 \log n) \Rightarrow d < 1$.
  - Vì $b^d < a$ (với $d < 1$ và $a = b = 2$)
- Theo Định lý Master, $T(n) = O(n^{\log_b a}) = O(n^{\log_2 2}) = O(n)$.

b)

Cho thuật toán minDistance với input gồm số nguyên dương $2 \leq n \leq 10^5$ và mảng số nguyên có các phần tử là số tự nhiên nhỏ hơn 10^9 như bên dưới:

```
int minDistance(int n, int a[]){
    int i = 0, res = 1e9;
    while(i < n){
        int j = i + 1;
        while(j < n){
            int cur = abs(a[i] - a[j]);
            if(cur < res) res = cur;
            j = j + 1;
        }
        i = i + 1;
    }
}
```

- Đánh giá độ phức tạp của thuật toán trên bằng cách đếm số phép so sánh $C(n)$
- Cho dù không tính được số phép gán $A(n)$ chính xác với mọi mảng input, ta vẫn có thể đánh giá được độ lớn của $A(n)$ theo ký hiệu Θ. Hỏi khẳng định này đúng không? Vì sao?
- Hãy mô tả ý tưởng một thuật toán khác cho ra cùng kết quả với thuật toán ở trên (dùng để tìm khoảng cách gần nhất giữa hai số hạng trong mảng) nhưng có độ phức tạp tốt hơn.

1. Đánh giá độ phức tạp của thuật toán bằng cách đếm số phép so sánh $C(n)$:

Trong thuật toán `minDistance`, chúng ta có hai vòng lặp lồng nhau:

- Vòng lặp ngoài chạy từ $i = 0$ đến $n - 1$.
- Vòng lặp trong chạy từ $j = i + 1$ đến $n - 1$.

Số lần lặp của vòng lặp ngoài là n, và mỗi lần lặp của vòng lặp ngoài, vòng lặp trong thực hiện $n - i$ phép lặp.

Tổng số phép so sánh $C(n)$ sẽ là:

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} (n - i - 1) = (n - 1) + (n - 2) + \ldots + 1 + 0 = \frac{n(n-1)}{2}$$

Vậy, độ phức tạp của thuật toán là $O(n^2)$.

2. Khẳng định đánh giá độ lớn của $A(n)$ theo ký hiệu Θ:

Khẳng định này đúng. Mặc dù không thể tính chính xác số phép gán $A(n)$ cho mọi input, nhưng chúng ta có thể đánh giá được độ lớn của $A(n)$ theo ký hiệu Θ dựa trên đánh giá tổng quan về số lần thực hiện phép gán trong thuật toán. Đối với một thuật toán có độ phức tạp $O(n^2)$, số phép gán $A(n)$ cũng sẽ có độ lớn tương tự, tức là $A(n) = \Theta(n^2)$.

Cụ thể:

- Trong trường hợp tốt nhất, điều kiện if(cur < res) xảy ra đúng 1 lần (tức phép gán trong `if` xảy ra đúng 1 lần), độ phức tạp của thuật toán được tính như sau:

$$A(n) = 2 + \left(\sum_{i=0}^{n-1} 2 + \left(\sum_{j=i+1}^{n-1} 2\right)\right) + 1 = 3 + \left(\sum_{i=0}^{n-1} 2 + 2(n-i-1)\right) = 3 + 2n + n(n-1) = n^2 + n + 3 \in \Omega(n$$

- Trong trường hợp xấu nhất, điều kiện if(cur < res) luôn xảy ra đúng 1 lần (tức phép gán trong `if` luôn xảy ra), độ phức tạp của thuật toán được tính như sau:

$$A(n) = 2 + \left(\sum_{i=0}^{n-1} 2 + \left(\sum_{j=i+1}^{n-1} 3\right)\right) = 2 + \left(\sum_{i=0}^{n-1} 2 + 3(n-i-1)\right) = 2 + 2n + \frac{3}{2}n(n-1) = \frac{3}{2}n^2 + \frac{1}{2}n + 2 \in O$$

- Như vậy thuật toán có độ phức tạp $\Theta(n^2)$

3. Thuật toán có độ phức tạp tốt hơn:

Sắp xếp mảng trước, sau đó chỉ cần tìm khoảng cách nhỏ nhất giữa các cặp phần tử liên tiếp trong mảng đã sắp xếp.

Cách thực hiện:

1. Sắp xếp mảng với độ phức tạp $O(n \log n)$.
2. Duyệt qua mảng đã sắp xếp để tìm khoảng cách nhỏ nhất giữa hai phần tử liên tiếp. Độ phức tạp của bước này là $O(n)$.

Tổng độ phức tạp của thuật toán sẽ là $O(n \log n)$, tốt hơn so với $O(n^2)$ của thuật toán ban đầu.

In [ ]:
```python
def minDistanceOptimized(arr):
    arr.sort()
    min_dist = float('inf')
    for i in range(1, len(arr)):
        min_dist = min(min_dist, arr[i] - arr[i-1])
    return min_dist
```

c)

Xét bài toán: "Cho hai chuỗi ký tự s_1, s_2 chỉ bao gồm các ký tự in hoa có cùng độ dài n $(1 \leq n \leq 10^5)$. Kiểm tra xem liệu một chuỗi có thể được tạo ra từ chuỗi còn lại bằng cách sắp xếp lại các ký tự hay không?". Chúng ta có hai ý tưởng để giải quyết vấn đề này:

1. Duyệt qua tất cả các ký tự của s_1, kiểm tra xem chúng có xuất hiện trong s_2 hay không, sau đó đánh dấu chúng là đã được kiểm tra. Nếu điều này đúng cho tất cả các ký tự của s_2, thì trả về `true`, ngược lại trả về `false`.

2. Sử dụng các biến phụ để đếm số lần xuất hiện của các ký tự 'A', 'B', ..., 'Z' trong s_1, s_2 và kiểm tra xem chúng có đều bằng nhau hay không.

Các yêu cầu:

- Triển khai các ý tưởng này bằng ngôn ngữ C++ hoặc Python.
- Sử dụng phương pháp thử nghiệm để đánh giá thời gian chạy của các thuật toán này tại 5 giá trị của n (từ nhỏ đến lớn): với mỗi giá trị n, tạo ngẫu nhiên một số chuỗi s_1, s_2 có độ dài đó để kiểm tra và tính toán thời gian chạy trung bình.

- Dựa trên phân tích ở phần trước, đưa ra một số phân tích về độ phức tạp của các thuật toán, chú ý đến sự đánh đổi giữa thời gian chạy và việc sử dụng bộ nhớ cho các biến phụ.
- Ngoài hai ý tưởng trên, hãy đưa ra thêm một ý tưởng nữa để giải quyết bài toán này?

In [ ]:
```python
def checkObtainedString1(s1, s2):
    visited = [-1]*len(s1)
    for i in range(len(s1)):
        for j in range(len(s2)):
            if s1[i] == s2[j] and visited[i] != j:
                visited[i] = j
                break
        if visited[i] == -1:
            return False
    return True
checkObtainedString1("abbd", "dbba")
```

Out[ ]:
```
True
```

- **Ý tưởng 1 (Kiểm tra ký tự xuất hiện):**
  - **Độ phức tạp thời gian:** $O(n^2)$ do sử dụng hai vòng lặp lồng nhau để kiểm tra từng ký tự trong s_1 và s_2.
  - **Sử dụng bộ nhớ:** $O(n)$ cho mảng `visited` để đánh dấu các ký tự đã kiểm tra.

In [ ]:
```python
def checkObtainedString2(s1, s2):
    count = [0]*26
    for i in range(len(s1)):
        count[ord(s1[i]) - ord('a')] += 1
    for i in range(len(s2)):
        count[ord(s2[i]) - ord('a')] -= 1
        if count[ord(s2[i]) - ord('a')] < 0:
            return False
    return True
checkObtainedString2("abbd", "dbba")
```

Out[ ]:
```
True
```

- **Ý tưởng 2 (Đếm số lượng ký tự):**
  - **Độ phức tạp thời gian:** $O(n)$ do chỉ cần duyệt qua các ký tự của s_1 và s_2 một lần.
  - **Sử dụng bộ nhớ:** $O(1)$ vì kích thước của mảng `count` là cố định (26 phần tử tương ứng với 26 chữ cái tiếng Anh).

In [ ]:
```python
def checkObtainedString3(s1, s2):
    sa = sorted(s1)
    sb = sorted(s2)
    return sa == sb
checkObtainedString3("abbd", "dbba")
```

Out[ ]:
```
True
```

- **Ý tưởng 3 (Sắp xếp chuỗi và so sánh):**
  - **Độ phức tạp thời gian:** $O(n \log n)$ do việc sắp xếp chuỗi.
  - **Sử dụng bộ nhớ:** $O(n)$ cho việc lưu trữ các chuỗi đã sắp xếp.

d)

Xây dựng mảng $a[1], a[2], \ldots, a[n]$ để lần lượt tính tổng các ước của các số $1, 2, \ldots, n$ theo các cách sau đây, với mỗi cách có giải thích rõ độ phức tạp:

- **Vét cạn $O(n^2)$.**
- **Sử dụng công thức để cải tiến $O(n\sqrt{n})$.**
- **Sử dụng ý tưởng sàng số nguyên tố $O(n \log n)$.**

1. Phương pháp vét cạn $O(n^2)$:

- **Ý tưởng:** Duyệt tất cả các ước của mỗi số từ 1 đến n và cộng tổng các ước đó.
- **Độ phức tạp:** Với mỗi i, có $O(i)$ phép tính, tổng cộng là $O(n^2)$.

In [ ]:
```
def sumOfDivisorsBruteforce(n):
    a = [0] * (n + 1)
    for i in range(1, n+1):
        for j in range(1, i+1):
            if i % j == 0:
                a[i] += j
    return a
sumOfDivisorsBruteforce(10)
```

Out[ ]:
```
[0, 1, 3, 4, 7, 6, 12, 8, 15, 13, 18]
```

2. Cải tiến với công thức $O(n\sqrt{n})$:

- **Ý tưởng:** Chỉ duyệt đến $\sqrt{i}$ để tìm ước, vì nếu j là ước của i thì $\frac{i}{j}$ cũng là ước.
- **Độ phức tạp:** Với mỗi i, có $O(\sqrt{i})$ phép tính, tổng cộng là $O(n\sqrt{n})$.

In [ ]:
```
import math
def sumOfDivisorsImproved(n):
    a = [0] * (n + 1)
    for i in range(1, n+1):
        for j in range(1, int(math.sqrt(i))+1):
            if i % j == 0:
                a[i] += j
                if j != i // j:
                    a[i] += i // j
    return a
sumOfDivisorsImproved(10)
```

Out[ ]:
```
[0, 1, 3, 4, 7, 6, 12, 8, 15, 13, 18]
```

3. Sàng số nguyên tố $O(n \log n)$:

- **Ý tưởng:** Cộng i vào tất cả các bội số của nó, giống như sàng nguyên tố.
- **Độ phức tạp:**
  - Với mỗi số i, ta cộng i vào tất cả các bội số của nó, nghĩa là ta thực hiện khoảng $\frac{n}{i}$ phép cộng.
  - Tổng số phép toán cho mọi i sẽ là:

$$O\left(\sum_{i=1}^{n} \frac{n}{i}\right) = O\left(n \sum_{i=1}^{n} \frac{1}{i}\right) = O(n \log n)$$

In [ ]:
```
def sumOfDivisorsSieve(n):
    a = [0] * (n + 1)
    for i in range(1, n+1):
        for j in range(i, n+1, i):
            a[j] += i
    return a
sumOfDivisorsSieve(10)
```

Out[ ]:
```
[0, 1, 3, 4, 7, 6, 12, 8, 15, 13, 18]
```

e)

Cho dãy số nguyên có n số $a_1, a_2, \ldots, a_n$ mà giá trị mỗi số thuộc tập hợp $\{0, 1, 2\}$. Cần đếm xem có tất cả mấy cách chọn ra cặp chỉ số (L, R) với $1 \leq L < R \leq n$ sao cho trong dãy con của a xét từ vị trí L đến vị trí R thì có một số nào đó xuất hiện từ 3 lần trở lên. Hãy mô tả ít nhất ba cách giải cho bài này với ba độ phức tạp khác nhau.

Ví dụ:

- $a = [0, 1, 2, 2, 2]$ thì đáp số là 3, ứng với 3 cách chọn cặp chỉ số: $(L, R) = (3, 5), (2, 5), (1, 5)$

Cách 1: Duyệt vét cạn 3 vòng for

- Ta duyệt theo L, duyệt theo R, kiểm tra khoảng từ $L \to R$ thì có thoả mãn không?
- Độ phức tạp: $O(n^3)$

In [ ]:
```
def countValid(a):
    n = len(a)
    res = 0
    for l in range(n):
        for r in range(l+1, n):
            count = [0]*3
            for k in range(l, r+1):
                count[a[k]] += 1
            if max(count) >= 3:
                res += 1
    return res
countValid([0,1,2,2,2])
```

Out[ ]: 3

Cách 2: Sử dụng mảng đếm

Ý tưởng:

- Sử dụng 3 mảng đếm để lưu số lượng các giá trị 0, 1, 2 xuất hiện trong dãy a đến vị trí i. Với mỗi cặp (L, R), kiểm tra điều kiện bằng cách sử dụng các giá trị đã lưu trong mảng đếm.

Cách làm:

- Giả sử $b(i)$, $c(i)$, $d(i)$ lần lượt là số lượng các giá trị 0, 1, 2 xuất hiện từ đầu đến vị trí i:

$$b(i) = b(i-1) + (a[i] == 0)$$

$$c(i) = c(i-1) + (a[i] == 1)$$

$$d(i) = d(i-1) + (a[i] == 2)$$

- Với mỗi cặp (L, R), điều kiện được kiểm tra bằng cách:

$$\text{if } b(R) - b(L-1) \geq 3 \text{ hoặc } c(R) - c(L-1) \geq 3 \text{ hoặc } d(R) - d(L-1) \geq 3$$

Độ phức tạp:

- Độ phức tạp của cách này là $O(n^2)$.

In [ ]:
```
def countValid1(a):
    n = len(a)
    res = 0
    prefixSum = [[0]*(n+1) for _ in range(3)]
    for i in range(1, n+1):
        prefixSum[0][i] = prefixSum[0][i-1] + (a[i-1] == 0)
        prefixSum[1][i] = prefixSum[1][i-1] + (a[i-1] == 1)
        prefixSum[2][i] = prefixSum[2][i-1] + (a[i-1] == 2)

    for l in range(1, n+1):
        for r in range(l+1, n+1):
            for k in range(3):
                if (prefixSum[k][r] - prefixSum[k][l-1]) >= 3:
                    res+=1
                    break
    return res
countValid1([0,1,2,2,2])
```

Out[ ]: 3

Cách 3: Áp dụng nguyên lý Dirichlet

Ý tưởng:

- Nếu đoạn con có độ dài lớn hơn hoặc bằng 7 thì theo nguyên lý Dirichlet, ít nhất sẽ có một số xuất hiện từ 3 lần trở lên. Chỉ cần kiểm tra các đoạn con có độ dài từ 7 trở lên.

Độ phức tạp:

- Độ phức tạp của cách này là $O(n)$.

In [ ]:
```python
def countValid2(a):
    n = len(a)
    res = 0
    prefixSum = [[0]*(n+1) for _ in range(3)]
    for i in range(1, n+1):
        prefixSum[0][i] = prefixSum[0][i-1] + (a[i-1] == 0)
        prefixSum[1][i] = prefixSum[1][i-1] + (a[i-1] == 1)
        prefixSum[2][i] = prefixSum[2][i-1] + (a[i-1] == 2)

    for l in range(1, n-1):
        for r in range(l+2, l+6):
            if r > n:
                break
            for k in range(3):
                if (prefixSum[k][r] - prefixSum[k][l-1]) >= 3:
                    res+=1
                    break
        if n - l > 5:
            res += n - (l+5)
    return res
countValid2([0,1,2,2,2])
```

Out[ ]: 3

Ghi chú: bài này có 1 phiên bản quen thuộc hơn là: đếm số dãy con mà trong đó có đủ 3 loại ký tự. Ta xây dựng 3 mảng đếm $b(i)$, $c(i)$, $d(i)$ như trên. Sau đó, ta cũng duyệt theo L rồi tìm kiếm nhị phân trên miền $b(i+1...n)$ xem chỉ số j đầu tiên nào mà $b(j) - b(i-1) >= 1$, nghĩa là có chứa số 0. Tương tự tìm chỉ số k để có $c(k) - c(i-1) \geq 1$ và chỉ số g để $d(g) - d(i-1) \geq 1$. Khi đó, vị trí đầu tiên thỏa mãn sẽ là: $max(j,k,g) \rightarrow res+= (n+1-max)$.

f)

Cho mảng a có n phần tử, mỗi phần tử là 0 hoặc 1. Cho phép chọn 1 dãy con liên tiếp và đổi trạng thái của chúng (0 sang 1 và 1 sang 0), làm ≤ 1 lần. Tìm tổng các số trong dãy mới thu được là lớn nhất có thể bằng các cách với độ phức tạp khác nhau.

Ví dụ: $n = 5$ và $a = [1,0,0,1,0]$; nếu đổi vị trí từ 2 đến $3 \rightarrow 1,1,1,1,0$ được tổng là 4; hoặc đổi từ 2 đến $5 \rightarrow 1,1,1,0,1$ vẫn tổng là 4.

Cách 1: Duyệt tất cả các dãy con

Ý tưởng: Xét tất cả các dãy con bằng cách sử dụng 2 vòng lặp cho vị trí `left` và vị trí `right` của dãy con.

Độ phức tạp: $O(n^3)$: Việc duyệt qua tất cả các dãy con với 2 vòng lặp lồng nhau dẫn đến độ phức tạp $O(n^2)$ để liệt kê các dãy con. Sau đó, với mỗi dãy con, ta lại tính tổng số `1`, dẫn đến độ phức tạp tổng cộng là $O(n^3)$.

In [ ]:
```python
def flip1(a):
    n = len(a)
    maxSum = sum(a)
    for l in range(n):
        for r in range(l, n):
            b = a.copy()
            for k in range(l, r+1):
                b[k] = abs(b[k]-1)
            maxSum = max(maxSum, sum(b))
    return maxSum
flip1([1,0,0,1,0])
```

Out[ ]: 4

Cách 2: Tính chênh lệch giữa số `0` và số `1`

Ý tưởng:

- Nhận xét: Nếu trong 1 dãy con đang thao tác, có a số `0` và b số `1` thì tổng hiện tại sẽ là b, sau thao tác thì tổng mới sẽ là a. Chênh lệch sẽ là `a-b`.

- Để cải tiến tốc độ, ta có thể tính trước số số `0` và số `1` (pre-compute) để biết số lượng số 0 và 1 từ vị trí `i` đến vị trí `j`.
- Sau đó, sử dụng thuật toán tối ưu để cập nhật kết quả một cách hiệu quả.

Độ phức tạp: $O(n^2)$: Việc tính toán chênh lệch và cập nhật kết quả chỉ cần một vòng lặp lồng nhau, nên độ phức tạp sẽ là $O(n^2)$.

In [ ]:
```
def flip2(a):
    n = len(a)
    maxSum = 0
    prefixSum = [[0]*(n+1) for _ in range(2)]
    for i in range(1, n+1):
        prefixSum[0][i] = prefixSum[0][i-1] + (a[i-1] == 0)
        prefixSum[1][i] = prefixSum[1][i-1] + (a[i-1] == 1)
    for l in range(1, n+1):
        for r in range(1, n+1):
            maxSum = max(maxSum, prefixSum[0][r] - prefixSum[0][l-1] - (prefixSum[1][r] - prefixSum[1][l
    return maxSum + sum(a)
flip2([1,0,0,1,0])
```

Out[ ]: 4

Cách 3: Sử dụng thuật toán Kadane (Maximum Subarray)

Ý tưởng:

- Chuyển bài toán về bài toán `Maximum Contiguous Subarray` (chuỗi con có tổng lớn nhất) bằng cách coi số `1` là `-1` và số `0` là `1`.
- Sau đó, áp dụng thuật toán `Kadane` để tìm dãy con liên tiếp có tổng lớn nhất.

Độ phức tạp: $O(n)$: Thuật toán `Kadane` chỉ cần một vòng lặp để tính tổng dãy con lớn nhất, nên độ phức tạp là $O(n)$.

In [ ]:
```
def flip3(a):
    n = len(a)
    b = [-1 if i == 1 else 1 for i in a]
    curMax = b[0]
    maxSoFar = b[0]
    for i in range(1, n):
        curMax = max(b[i], b[i] + curMax)
        maxSoFar = max(maxSoFar, curMax)
    return sum(a) + maxSoFar
flip3([1,0,0,1,0])
```

Out[ ]: 4

g)

Đếm số cách chia mảng độ dài $n \leq 5 * 10^5$ thành 2 phần khác rỗng, gồm các số $a[i]$ mà $|a[i]| \leq 10^9$ và có tổng bằng nhau.

Ý tưởng:

- Sử dụng mảng `prefixSum` để lưu trữ tổng các phần tử từ đầu đến vị trí i.
- Tìm xem có bao nhiêu vị trí trong mảng mà tổng từ đầu đến đó bằng một nửa tổng của toàn mảng.
- Kết quả là số lần tìm được tổng này.

Độ phức tạp:

- Độ phức tạp thời gian: $O(n)$
- Độ phức tạp không gian: $O(n)$

In [ ]:
```
def numberOfWays(a):
    n = len(a)
    S = sum(a)
    if S%2 != 0:
        return 0
```

```
    S = S//2
    count = 0
    prefixSum = [0] * (n+1)
    for i in range(1, n+1):
        prefixSum[i] = prefixSum[i-1] + a[i-1]
        if prefixSum[i] == S:
            count += 1
    return count
numberOfWays([1,2,3,0,0,6])
```

Out[ ]: 3

Đếm số cách chia mảng độ dài $n \leq 5 * 10^5$ thành 3 phần khác rỗng, gồm các số $a[i]$ mà $|a[i]| \leq 10^9$ và có tổng bằng nhau.

VD:

- n = 5 và 1,2,3,0,3 ⇒ [1,2],[3],[0,3] hoặc [1,2],[3,0],[3] ⇒ 2 cách.
- n = 4 và 0,0,0,0 ⇒ [0],[0],[0,0]; [0,0],[0],[0]; [0],[0,0],[0]; [0,0,0],[0]; [0],[0],[0],[0]; ⇒ 3 cách.
- n = 3 và 1,2,3 ⇒ không thể chia được ⇒ 0 cách.

Ý tưởng: Để tìm số cách chia mảng thành 3 phần có tổng bằng nhau, trước hết ta cần tính tổng của mảng `S`. Nếu `S` không chia hết cho 3, thì rõ ràng không thể chia mảng thành 3 phần có tổng bằng nhau và trả về 0.

- Nếu `S` chia hết cho 3, ta tiến hành duyệt mảng để tính tổng tiền tố `prefixSum`. Nếu `prefixSum` tại vị trí `i` bằng `S/3`, ta tìm các vị trí tiếp theo từ `i+1` đến `n` sao cho tổng từ `i+1` đến `j` cũng bằng `S/3`, nếu có thì ta tăng biến đếm.

Độ phức tạp:

- Cách này duyệt qua mảng hai lần, với mỗi phần tử `i` có thể duyệt hết từ `i+1` đến `n`, do đó độ phức tạp thời gian là `O(n^2)`.

In [ ]:
```
def numberOfWays(a):
    n = len(a)
    S = sum(a)
    if S%3 != 0:
        return 0
    S = S//3
    count = 0
    prefixSum = [0] * (n+1)
    for i in range(1, n+1):
        prefixSum[i] = prefixSum[i-1] + a[i-1]
    for i in range(1, n+1):
        if prefixSum[i] == S:
            for j in range(i+1, n):
                if prefixSum[j] - prefixSum[i] == S:
                    count+=1
    return count
numberOfWays([1,2,3,0,3])
```

Out[ ]: 2

Ý tưởng: Thay vì tìm tất cả các phần tử `j` từ `i+1` đến `n` để tạo ra phần tử thứ 2 có tổng `S/3`, ta dùng một mảng `sufixCount` để lưu trữ số lượng cách tạo ra tổng `S/3` từ `i+1` đến `n` và sử dụng mảng này để tính nhanh số cách chia.

- **Độ phức tạp:**
  - Cách này chỉ cần duyệt qua mảng 3 lần, với mỗi lần duyệt có độ phức tạp `O(n)`. Do đó, tổng độ phức tạp của cách này là `O(n)`.

In [ ]:
```
def numberOfWays(a):
    n = len(a)
    S = sum(a)
    if S%3 != 0:
        return 0
    S = S//3
    count = 0
    prefixSum = [0] * (n+1)
```

```
    sufixCount = [0] * (n+1)

    for i in range(1, n+1):
        prefixSum[i] = prefixSum[i-1] + a[i-1]

    for i in range(n-1, -1, -1):
        sufixCount[i] = sufixCount[i+1] + (prefixSum[n] - prefixSum[i] == S)

    for i in range(1, n):
        if prefixSum[i] == S:
            count += sufixCount[i+1]

    return count
numberOfWays([1,2,3,0,3])
```

Out[ ]: 2

Bài 5

a)

Cho hai số nguyên dương n, m với $m \leq 10^{14}$, $n \leq 10^5$. Hỏi có thể biểu diễn n theo các ước của m với ít nhất mấy số hạng?

- VD:
  - $n = 7, m = 12 \Rightarrow kq = 2$ vì $7 = 3 + 4$
  - $n = 7, m = 8 \Rightarrow kq = 3$ vì $7 = 1 + 2 + 4$

Lưu ý: Câu hỏi tương tự nếu thay bởi số nguyên tố, số chính phương, số Fibonacci.

Ý tưởng:

- Tìm tất cả các ước của m.
- Sử dụng mảng $dp[i]$ để lưu trữ số lượng số hạng ít nhất cần thiết để tạo thành tổng i.
- Công thức quy hoạch động:

$$dp[i] = \min\{dp[i - \text{divisor}[j]] + 1\} \quad \text{với} \quad i \geq \text{divisor}[j]$$

- Duyệt qua tất cả các ước $\text{divisor}[j]$ của m và cập nhật giá trị của dp cho các tổng từ 1 đến n.

Độ phức tạp của thuật toán:

- **Thời gian:**
  - Tìm các ước của m: $O(\sqrt{m})$.
  - Tính toán với Dynamic Programming: $O(n \times d)$ với d là số lượng ước của m.
  - Tổng độ phức tạp: $O(\sqrt{m}) + O(n \times d)$.
- **Không gian:**
  - Sử dụng mảng dp có độ phức tạp không gian $O(n)$.

In [ ]:
```
def getDivisors(n):
    res = []
    for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
            res.append(i)
            if i != n // i:
                res.append(n // i)
    return sorted(res)

def minDivisors(n, m):
    divisorList = getDivisors(m)
    dp = [n] * (n+1)
    dp[0] = 0
    for i in range(1, n+1):
        for j in range(len(divisorList)):
            if i >= divisorList[j]:
```

```
                dp[i] = min(dp[i], dp[i - divisorList[j]] + 1)
    return dp[n]
minDivisors(7, 8)
```

Out[ ]: 3

b)

Cho chuỗi s gồm các ký tự tiếng Anh in thường độ dài n. Hỏi có thể cắt s thành được nhiều nhất mấy chuỗi con `không` đối xứng (palidrom)?

- VD:
  - $aab \rightarrow 1$
  - $ababcd \rightarrow 3$, chẳng hạn: ab, ab, cd

Chú ý: Chuỗi có độ dài 1 vẫn tính là đối xứng

In [ ]:
```
def maxNonPalindrom(s):
    n = len(s)
    dp = [0] * n
    dp[0] = 0
    dp[1] = 1 if s[0] != s[1] else 0
    for i in range(2, n):
        if s[i] == s[i-1]:
            dp[i] = dp[i-1]
        else:
            dp[i] = dp[i-2] + 1
    return dp[n-1]
maxNonPalindrom("ababcd")
```

Out[ ]: 3

Giải thích ý tưởng:

- **Ý tưởng chính:**
  - Sử dụng mảng $dp[i]$ để lưu trữ số lượng chuỗi con không đối xứng tối đa mà chuỗi $s[0:i]$ có thể được chia thành.
  - Khởi tạo:
    - $dp[0] = 0$ vì không có chuỗi con trong chuỗi rỗng.
    - $dp[1] = 1$ nếu $s[0] \neq s[1]$, ngược lại $dp[1] = 0$.
  - Duyệt qua chuỗi từ vị trí 2 đến $n-1$:
    - Nếu hai ký tự liên tiếp $s[i]$ và $s[i-1]$ giống nhau, nghĩa là chuỗi con $s[i-1:i]$ là palindrome, số lượng chuỗi con không đối xứng tối đa tính đến vị trí i chính là $dp[i-1]$
    - Nếu hai ký tự $s[i]$ và $s[i-1]$ khác nhau, nghĩa là chuỗi con $s[i-1:i]$ không là palindrome, số lượng chuỗi con không đối xứng tối đa tính đến vị trí i chính là $dp[i-2]+1$
- **Công thức quy hoạch động tổng quát:**

$$dp[i] = \begin{cases} dp[i-1] & \text{nếu } s[i] = s[i-1] \\ dp[i-2]+1 & \text{nếu } s[i] \neq s[i-1] \end{cases}$$

Phân tích độ phức tạp:

- **Thời gian:** $O(n)$
  - Thuật toán chỉ cần duyệt qua chuỗi s một lần để tính toán giá trị của mảng dp.
- **Không gian:** $O(n)$
  - Thuật toán sử dụng một mảng dp để lưu trữ số lượng chuỗi con không đối xứng tại mỗi vị trí i trong chuỗi.

c)

Phân hoạch Palindrome

Cho một chuỗi str, một phân hoạch của chuỗi này là một phân hoạch palindrome nếu mọi chuỗi con trong phân hoạch đều là palindrome. Nhiệm vụ là tìm cách phân hoạch chuỗi str sao cho số lần cắt là ít nhất và tất cả các chuỗi con thu được đều là palindrome.

Ví dụ:

1. **Ví dụ 1:**
   - Đầu vào: $\text{str} = aab$
   - Đầu ra: 1
   - Giải thích: Ta có thể cắt sau ký tự thứ 2 để được hai chuỗi con "aa" và "b". "aa" là palindrome và "b" cũng là palindrome. Số lần cắt cần thiết là 1.
2. **Ví dụ 2:**
   - Đầu vào: $\text{str} = banana$
   - Đầu ra: 1
   - Giải thích: Ta có thể cắt sau ký tự đầu tiên để được hai chuỗi con "b" và "anana". Cả hai đều là palindrome. Số lần cắt cần thiết là 1.
3. **Ví dụ 3:**
   - Đầu vào: $\text{str} = racecar$
   - Đầu ra: 0
   - Giải thích: Cả chuỗi "racecar" là một palindrome, do đó không cần cắt.

Ý tưởng chính:

- Sử dụng mảng `dp[i]` để lưu trữ số lần cắt tối thiểu cần thiết để phân hoạch chuỗi từ vị trí 0 đến i.
- Sử dụng bảng `palindrome` để kiểm tra nhanh xem một chuỗi con $\text{str}[j:i]$ có phải là palindrome hay không.

Công thức quy hoạch động tổng quát:

$$dp[i] = \min(dp[i], dp[j-1] + 1) \quad \text{với mọi } j \leq i \quad \text{và } \text{str}[j:i] \text{ là palindrome}$$

- Nếu $\text{str}[0:i]$ là palindrome, thì $dp[i] = 0$ (không cần cắt).
- Cách kiểm tra chuỗi $\text{str}[j:i]$ có phải là chuỗi palidrom hay không:
  - $\text{str}[j:i]$ là chuỗi Palindrom nếu $\text{str}[i] = \text{str}[j]$ và $\text{str}[j+1:i-1]$ là 1 chuỗi Palindrom.

Phân tích độ phức tạp:

- **Thời gian:** $O(n^2)$
  - Có hai vòng lặp lồng nhau để duyệt qua các vị trí i và j nhằm tính toán số lần cắt tối thiểu.
  - Kiểm tra một chuỗi con có phải là palindrome hay không sử dụng bảng `palindrome`, mỗi lần kiểm tra tốn $O(1)$ sau khi bảng được xây dựng.
- **Không gian:** $O(n^2)$
  - Mảng `dp` có kích thước n và bảng `palindrome` có kích thước $n \times n$ để lưu trữ thông tin.

In [ ]:
```python
def palindromPartioning(s):
    n = len(s)
    dp = list(range(n))
    palidrom = [[False]*n for _ in range(n)]
    for i in range(n):
        for j in range(i+1):
            if (s[i] == s[j]) and ((i - j < 2) or palidrom[j+1][i-1]):
                palidrom[j][i] = True
                if j == 0:
                    dp[i] = 0
                    break
                dp[i] = min(dp[i], dp[j-1] + 1)
    return dp[n-1]
palindromPartioning("abcde")
```

Out[ ]: 4

d)

Cho trước các số n, m, k với $mk \leq n$. Từ một mảng có n phần tử không âm và các số nguyên dương, hãy xây dựng công thức quy hoạch động để xác định tổng lớn nhất trong k mảng con rời nhau, mỗi mảng có đúng m phần tử liên tiếp.

Ví dụ:

- $n = 5$ và $1, 2, 3, 4, 5$ và $k = 2$; lấy ra mảng $[2, 3]$ và $[4, 5] \Rightarrow 2 + 3 + 4 + 5 = 14$.
- $n = 5$ và $5, 2, 6, 3, 4$ và $k = m = 2 \Rightarrow [5, 2]$ và $[6, 3] \Rightarrow 5 + 2 + 6 + 3 = 16$.
- $n = 5$ và $5, 2, 6, 3, 4$ và $k = 1$, $m = 3 \Rightarrow [5, 2, 6]$ hoặc $[6, 3, 4] \Rightarrow$ tổng max = 13.
- $n = 5$ và $5, 2, 6, 3, 4$ và $k = 3$, $m = 1 \Rightarrow [5]$, $[6]$, $[4] \Rightarrow$ tổng max = 15.

Ý tưởng chính:

- Sử dụng quy hoạch động để tính toán tổng lớn nhất cho mỗi số lượng mảng con từ 1 đến k.
- Dùng mảng `prefixSum` để tính tổng của bất kỳ đoạn con nào một cách nhanh chóng.
- Mảng `dp[i][j]` lưu trữ tổng lớn nhất có thể khi chọn j mảng con từ các phần tử đầu tiên đến phần tử thứ i.

Công thức quy hoạch động tổng quát:

$$dp[i][j] = \max\left(dp[i-1][j], dp[i-m][j-1] + \text{sum}(a[i-m+1 \ldots i])\right)$$

- Trong đó:
  - $dp[i-1][j]$: Tổng lớn nhất khi chọn j mảng con từ $i-1$ phần tử.
  - $dp[i-m][j-1] + \text{sum}(a[i-m+1 \ldots i])$: Tổng lớn nhất khi chọn $j-1$ mảng con từ $i-m$ phần tử đầu tiên cộng với tổng của mảng con mới $a[i-m+1 \ldots i]$.
  - Với $\text{sum}(a[i-m+1 \ldots i]) = \text{prefixSum}[i] - \text{prefixSum}[i-m]$

Phân tích độ phức tạp:

- **Thời gian:** $O(n \times k)$
  - Có hai vòng lặp chính: vòng lặp i từ m đến n và vòng lặp j từ 1 đến k.
  - Mỗi phép tính trong vòng lặp tốn $O(1)$, nhưng tính toán `prefixSum` ban đầu tốn $O(n)$.
- **Không gian:** $O(n \times k)$
  - Cần sử dụng mảng `dp` kích thước $(n+1) \times (k+1)$ để lưu trữ kết quả của từng trạng thái.

In [ ]:
```python
def max_sum_subarrays(a, m, k):
    n = len(a)
    dp = [[0]*(k+1) for _ in range(n+1)]
    prefixSum = [0] * (n+1)

    for i in range(1, n+1):
        prefixSum[i] = prefixSum[i-1] + a[i-1]

    for i in range(m, n+1):
        for j in range(1, k+1):
            dp[i][j] = max(dp[i-1][j], dp[i-m][j-1] + prefixSum[i] - prefixSum[i-m])

    return dp[n][k]
a = [5,2,6,3,4]
m, k = 1, 3
max_sum_subarrays(a, m, k)
```

Out[ ]:
```
15
```

e)

Bài toán cái túi (Knapsack Problem)

Cho một cái túi có thể chứa tối đa trọng lượng W. Có n đồ vật, mỗi đồ vật thứ i có trọng lượng $w[i]$ và giá trị $v[i]$

Nhiệm vụ là chọn một tập hợp các đồ vật để đưa vào túi sao cho tổng trọng lượng của các đồ vật không vượt quá W và tổng giá trị của chúng là lớn nhất.

Ví dụ:

- Đầu vào: $W = 20$, $n = 5$, $w = [10, 4, 9, 6, 7]$, $v = [5, 2, 4, 6, 1]$
- Đầu ra: 13

Ý tưởng chính:

- Mảng `dp[i][j]` lưu trữ giá trị lớn nhất có thể đạt được khi chọn từ 1 đến j đồ vật với tổng trọng lượng không vượt quá i.

Công thức quy hoạch động tổng quát:

$$dp[i][j] = \max\left(dp[i][j-1], dp[i-w[j-1]][j-1] + v[j-1]\right) \quad \text{với } i \geq w[j-1]$$

- Trong đó:
  - `dp[i][j-1]` : Giá trị lớn nhất có thể đạt được khi không chọn đồ vật thứ j.
  - `dp[i-w[j-1]][j-1] + v[j]` : Giá trị lớn nhất có thể đạt được khi chọn đồ vật thứ j và thêm giá trị của nó vào kết quả của các đồ vật trước đó.

Phân tích độ phức tạp:

- **Thời gian:** $O(n \times W)$
  - Có hai vòng lặp chính: vòng lặp đầu tiên từ 1 đến W (độ phức tạp $O(W)$) và vòng lặp thứ hai qua các đồ vật (độ phức tạp $O(n)$).
- **Không gian:** $O(n \times W)$
  - Cần sử dụng mảng `dp` kích thước $(W+1) \times (n+1)$ để lưu trữ kết quả của từng trạng thái.

In [ ]:
```python
def knapsackProblem(C, n, W, V):
    dp = [[0]*(n+1) for _ in range(C+1)]
    for i in range(1, C+1):
        for j in range(1, n+1):
            dp[i][j] = dp[i][j-1]
            if i >= W[j-1]:
                dp[i][j] = max(dp[i][j], dp[i-W[j-1]][j-1] + V[j-1])
    return dp[C][n]
C = 20
n = 5
W = [10, 4, 9, 6, 7]
V = [5, 2, 4, 6, 1]
knapsackProblem(C, n, W, V)
```

Out[ ]:
```
13
```

f)

Bài toán đổi tiền xu

Gọi mệnh giá k đồng xu là $\{x_1, x_2, \ldots, x_k\}$ và giả định rằng, đồng xu có mệnh giá nhỏ nhất là 1 xu để đảm bảo luôn có lời giải.

Ý tưởng chính:

- Mảng `dp[i]` lưu trữ số lượng đồng xu ít nhất để đổi được i xu.

Công thức quy hoạch động tổng quát:

$$dp[i] = \min_{1 \leq j \leq k} \{dp[i - x_j] + 1\} \quad \text{với } i \geq x_j$$

Phân tích độ phức tạp:

- **Thời gian:** $O(n \times k)$

- Có hai vòng lặp chính: vòng lặp đầu tiên từ 1 đến n (độ phức tạp $O(n)$) và vòng lặp thứ hai qua các mệnh giá (độ phức tạp $O(k)$).

- **Không gian:** $O(n)$
  - Cần sử dụng mảng `dp` kích thước $n+1$ để lưu trữ kết quả của từng giá trị từ 0 đến n.

In [ ]:
```python
def changeCoinDP(deno, n, k):
    dp = [0] * (k+1)
    for i in range(1, k+1):
        dp[i] = n+1
        for j in range(n):
            if i >= deno[j]:
                dp[i] = min(dp[i], dp[i-deno[j]] + 1)
    return dp[k]
deno = [1, 5, 10, 25, 50]
n, k = 5, 72
changeCoinDP(deno, n, k)
```

Out[ ]: 5

g)

Lavrenty có n gam bột cùng với m loại nhân khác nhau. Các loại nhân được đánh số từ 1 đến m. Lavrenty biết rằng anh ta có a_i gam của loại nhân thứ i. Để làm một chiếc bánh với nhân thứ i, cần chính xác b_i gam nhân và c_i gam bột. Một chiếc bánh như vậy có thể bán với giá d_i tugrik.

Ngoài ra, anh ta cũng có thể làm bánh không nhân. Mỗi chiếc bánh như vậy cần c_0 gam bột và có thể bán với giá d_0 tugrik. Vì vậy Lavrenty có thể nướng bất kỳ số lượng bánh nào với nhân khác nhau hoặc không nhân trừ khi anh ta hết bột và nhân. Lavrenty sẽ bỏ hết nguyên liệu còn lại sau khi nướng.

Nhiệm vụ của bạn là tìm số lượng tugrik tối đa mà Lavrenty có thể kiếm được.

Ví dụ: $n = 10, m = 2, c_0 = 2, d_0 = 1, a = [7, 12], b = [3, 3], c = [2, 1], d = [100, 10] \Rightarrow$ Kết quả là: 241

Ý tưởng chính:

- Sử dụng quy hoạch động để tính toán giá trị lớn nhất có thể kiếm được cho mỗi lượng bột từ 0 đến n.
- Mảng `dp[i][j]` lưu trữ giá trị lớn nhất có thể kiếm được khi sử dụng i gam bột và j loại nhân khác nhau.

Công thức quy hoạch động tổng quát:

$$dp[i][j] = \max\left(dp[i][j-1], dp[i - k \times c_j][j-1] + k \times d_j\right) \quad \text{với } (i \geq k \times c_j) \wedge (a_j \geq k \times b_j)$$

- Trong đó:
  - $dp[i][j-1]$: Giá trị lớn nhất có thể đạt được khi không sử dụng loại nhân thứ j.
  - $dp[i - k \times c_j][j-1] + k \times d_j$: Giá trị lớn nhất có thể đạt được khi sử dụng k lần loại nhân thứ j và thêm giá trị của nó vào kết quả của các loại nhân trước đó.

Kết quả cuối cùng:

Ta cần cộng thêm lợi nhuận của phần bánh không cần dùng nhân vào kết quả nếu dư $n - i$ lượng bột.

$$\max\left(dp[i][m] + d_0 * \left\lfloor \frac{n-i}{c_0} \right\rfloor\right) \quad \text{với } 0 \leq i \leq n$$

Phân tích độ phức tạp:

- **Thời gian:** $O(n \times m \times k)$
  - Có ba vòng lặp chính: vòng lặp đầu tiên từ 1 đến n (độ phức tạp $O(n)$), vòng lặp thứ hai qua các loại nhân (độ phức tạp $O(m)$), và vòng lặp thứ ba xét số lần k (độ phức tạp $O(k)$).
- **Không gian:** $O(n \times m)$
  - Cần sử dụng mảng `dp` kích thước $(n+1) \times (m+1)$ để lưu trữ kết quả của từng trạng thái.

In [ ]:
```python
def buns(n, m, c0, d0, a, b, c, d):
    dp = [[0]*(m+1) for _ in range(n+1)]

    for i in range(1, n+1):
        dp[i][0] = d0 * (i//c0)
        for j in range(1, m+1):
            dp[i][j] = dp[i][j-1]
            for k in range(i//c[j-1]):
                if a[j-1] >= k*b[j-1]:
                    dp[i][j] = max(dp[i][j], dp[i-k*c[j-1]][j-1] + k*d[j-1])
    res = dp[n][m]
    for i in range(n+1):
        res = max(res, dp[i][m] + d0*((n-i)//c0))
    return res
n, m, c0, d0 = 10, 2, 2, 1
a = [7, 12]
b = [3, 3]
c = [2, 1]
d = [100, 10]
buns(n, m, c0, d0, a, b, c, d)
```

Out[ ]: 241

h)

Hỏi có mấy cách đi từ $(0, 0)$ đến (m, n) sao cho mỗi bước chỉ được $\uparrow, \nearrow, \rightarrow$

Ý tưởng chính:

- Mảng `dp[i][j]` lưu trữ số cách đi đến ô (i, j).

Công thức quy hoạch động tổng quát:

$$dp[i][j] = dp[i-1][j] + dp[i][j-1] + dp[i-1][j-1]$$

- Trong đó:
  - `dp[i-1][j]` : Số cách đi đến ô (i, j) từ ô bên dưới.
  - `dp[i][j-1]` : Số cách đi đến ô (i, j) từ ô bên trái.
  - `dp[i-1][j-1]` : Số cách đi đến ô (i, j) từ ô chéo dưới bên trái.

Phân tích độ phức tạp:

- **Thời gian:** $O(m \times n)$
  - Có hai vòng lặp chính để tính toán số cách đi cho mỗi ô trên lưới 2D với m hàng và n cột.
- **Không gian:** $O(m \times n)$
  - Cần sử dụng mảng `dp` kích thước $(m+1) \times (n+1)$ để lưu trữ số cách đi cho mỗi trạng thái.

In [ ]:
```python
def numberOfWays(x, y):
    dp = [[0]*(x+1) for _ in range(y+1)]
    for i in range(x+1):
        dp[0][i] = 1
    for i in range(y+1):
        dp[i][0] = 1
    for i in range(1, y+1):
        for j in range(1, x+1):
            dp[i][j] = dp[i-1][j] + dp[i][j-1] + dp[i-1][j-1]
    return dp[y][x]
numberOfWays(5, 10)
```

Out[ ]: 36365

i)

Hỏi có mấy cách để bước lên bậc cầu thang n bậc mà mỗi bước cho phép bước 1, 2 hoặc 3 bước?

Ý tưởng chính:

- Mảng `dp[i]` lưu trữ số cách để lên đến bậc thang thứ i.

Công thức quy hoạch động tổng quát:

$$dp[i] = dp[i-1] + dp[i-2] + dp[i-3]$$

Phân tích độ phức tạp:

- **Thời gian:** $O(n)$
  - Có một vòng lặp đơn từ 4 đến n để tính toán số cách bước lên mỗi bậc, do đó độ phức tạp thời gian là $O(n)$.
- **Không gian:** $O(n)$
  - Cần sử dụng mảng `dp` kích thước $n+1$ để lưu trữ số cách bước lên mỗi bậc thang.

In [ ]:
```python
def numberOfStep(n):
    dp = [0] * (n+1)
    dp[1] = 1
    dp[2] = 2
    dp[3] = 4
    for i in range(4, n+1):
        dp[i] = dp[i-1] + dp[i-2] + dp[i-3]
    return dp[n]
numberOfStep(5)
```

Out[ ]:
```
13
```

k)

Bài toán LIS (Largest Increasing Subsequence). Cho mảng gồm n số $a_1, a_2, \ldots, a_n$. Tìm dãy con sao cho: $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$ với k lớn nhất.

Ý tưởng chính:

- Mảng `dp[i]` lưu trữ chiều dài của dãy con tăng dần dài nhất kết thúc tại phần tử thứ i.

Công thức quy hoạch động tổng quát:

$$dp[i] = \max(dp[j] + 1) \quad \text{với mọi } j < i \text{ và } a[j] < a[i]$$

- Trong đó:
  - `dp[j]` : Chiều dài của dãy con tăng dần dài nhất kết thúc tại phần tử thứ j.
  - Điều kiện `a[j] < a[i]` đảm bảo rằng dãy con tiếp tục tăng dần khi thêm phần tử $a[i]$.

Phân tích độ phức tạp:

- **Thời gian:** $O(n^2)$
  - Có hai vòng lặp lồng nhau: vòng lặp ngoài chạy từ 1 đến n, và vòng lặp trong chạy từ 0 đến $i-1$. Do đó độ phức tạp thời gian là $O(n^2)$.
  - Ta có thể cải tiến bước duyệt tính $dp[i]$ ở trên nhờ việc kiểm soát các giá trị trong kết quả bởi 1 mảng bổ sung rồi tìm kiếm nhị phân trên đó để có chi phí là $O(n \log n)$
- **Không gian:** $O(n)$
  - Cần sử dụng mảng `dp` kích thước n để lưu trữ chiều dài của dãy con tăng dần dài nhất tại mỗi phần tử.

In [ ]:
```python
def LIS(a):
    n = len(a)
    dp = [1] * n
    for i in range(1, n):
        dp[i] = dp[i-1]
        for j in range(i):
            if a[i] > a[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return dp[n-1]
LIS([50, 3, 10, 7, 40, 80])
```

Out[ ]: 4

I)

Cho mảng $a[1...n]$ gồm phần tử 0, 1. Đếm số cặp L, R sao cho: trong mảng con $a[L...R]$, các cụm số 0 liên tục có số lượng số 0 là chia hết cho k.

Sử dụng Brute-Force:

- Sử dụng hai vòng lặp để duyệt qua tất cả các cặp chỉ số L và R trong mảng a. Với mỗi đoạn con $a[L...R]$, hàm `check_clusters` sẽ kiểm tra xem các cụm số 0 trong đoạn này có số lượng chia hết cho k hay không.

Phân tích độ phức tạp

- **Thời gian:** $O(n^3)$
  - Có hai vòng lặp L và R chạy từ 0 đến n, mất $O(n^2)$ thời gian.
  - Với mỗi đoạn con $a[L...R]$, hàm `check_clusters` có thể cần duyệt toàn bộ đoạn con, mất $O(n)$ thời gian.
  - Tổng thời gian chạy của thuật toán là $O(n^3)$.
- **Không gian:** $O(1)$
  - Chỉ sử dụng thêm các biến phụ để kiểm tra các đoạn con và các cụm số 0.

Thuật toán này có độ phức tạp thời gian lớn, không tối ưu cho các mảng có kích thước lớn.

In [ ]:
```python
def check_clusters(arr, L, R, k):
    i = L
    while i <= R:
        if arr[i] == 0:
            cluster_size = 0
            while i <= R and arr[i] == 0:
                cluster_size += 1
                i += 1
            if cluster_size % k != 0:
                return False
        else:
            i += 1
    return True

def count_valid_subarrays_bruteforce(a, n, k):
    result = 0
    for L in range(n):
        for R in range(L, n):
            if check_clusters(a, L, R, k):
                result += 1
    return result
count_valid_subarrays_bruteforce([1, 0, 0, 1, 0, 0, 0, 0, 1], len(a), 2)
```

Out[ ]: 1

Công thức quy hoạch động

- Ta sử dụng mảng `dp[i]` để lưu số lượng đoạn con hợp lệ kết thúc tại vị trí i.

Công thức tổng quát:

$$dp[i] = \begin{cases} dp[i-1] + 1 & \text{nếu } a[i] = 1 \\ dp[i-k] + 1 & \text{nếu } a[i] = a[i-1] = a[i-2] = \ldots = a[i-k+1] = 0 \\ 0 & \text{ngược lại} \end{cases}$$

Phân tích độ phức tạp

- **Thời gian:** $O(n \times k)$ do thuật toán duyệt qua mảng `a` một lần ($O(n)$) và với mỗi phần tử `0`, kiểm tra `k` phần tử trước đó.
- **Không gian:** $O(n)$ do sử dụng mảng `dp` kích thước `n` để lưu trữ trạng thái.

Thuật toán có hiệu suất tốt khi n và k không quá lớn.

In [ ]:
```python
def countClusterZerosDP(a, k):
    n = len(a)
    dp = [0]*n
    dp[0] = 1 if a[0] == 1 else 0 if k != 1 else 1
    for i in range(1, n):
        if a[i] == 1:
            dp[i] = dp[i-1] + 1
        else:
            flag = True
            for j in range(k):
                if a[i-1+j] != 0:
                    dp[i] = 0
                    flag = False
                    break
            if flag:
                dp[i] = dp[i-k] + 1
    return sum(dp)
countClusterZerosDP([1, 0, 0, 1, 0, 0, 0, 0, 1], 2)
```

Out[ ]: 22

m)

Đếm số cách lát nền nhà $3 \times n$ bằng các viên gạch kích thước 1×3 hoặc 3×1

Ý tưởng:

- Sử dụng quy hoạch động để đếm số cách lát nền nhà.
- Gọi $dp[i]$ là số cách để lát nền nhà kích thước $3 \times i$.
- Cách lát nền:
  - Nếu ta đặt viên gạch 1×3 theo chiều dọc, ta chỉ cần lát phần còn lại là $3 \times (i-1)$.
  - Nếu ta đặt viên gạch 3×1 theo chiều ngang, ta cần lát phần còn lại là $3 \times (i-3)$.

Công thức quy hoạch động tổng quát

$$dp[i] = dp[i-1] + dp[i-3]$$

- Với:
  - $dp[1] = 1$
  - $dp[2] = 2$
  - $dp[3] = 2$

Phân tích độ phức tạp

- **Thời gian:** $O(n)$ - Thuật toán duyệt từ 1 đến n để tính toán số cách lát nền.
- **Không gian:** $O(n)$ - Sử dụng mảng dp có kích thước $n+1$ để lưu trữ kết quả.

In [ ]:
```python
def numberOfWays(n):
    dp = [0] * (n+1)
    dp[1] = 1
    dp[2] = 2
    dp[3] = 2
    for i in range(4, n+1):
        dp[i] = dp[i-1] + dp[i-3]
    return dp[n]
numberOfWays(10)
```

Out[ ]: 32

n)

Cho 2 chuỗi ký tự là s_1 và s_2. Tính độ dài chuỗi con chung dài nhất (không nhất thiết liên tiếp) của s_1 và s_2 (bài toán LCS)

Ý tưởng:

- Gọi $dp[i][j]$ là độ dài chuỗi con chung dài nhất của chuỗi s_1 có độ dài i và chuỗi s_2 có độ dài j.
- Nếu $s_1[i] = s_2[j]$ thì ta có thể tăng độ dài chuỗi con chung lên 1
- Ngược lại, nếu $s_1[i] \neq s_2[j]$, ta sẽ chọn giá trị lớn nhất từ hai trường hợp bỏ đi một ký tự ở s_1 hoặc s_2

Công thức quy hoạch động tổng quát:

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1 & \text{nếu } s_1[i] = s_2[j] \\ \max(dp[i-1][j], dp[i][j-1]) & \text{ngược lại} \end{cases}$$

- Khởi tạo: $dp[0][j] = 0$ và $dp[i][0] = 0$.

Phân tích độ phức tạp:

- **Thời gian:** $O(m \times n)$, với m là độ dài chuỗi s_1 và n là độ dài chuỗi s_2.
- **Không gian:** $O(m \times n)$ do sử dụng bảng dp có kích thước $(m+1 \times n+1)$.

In [ ]:
```
def LCS(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0]*(m + 1) for _ in range(n + 1)]
    for i in range(1, n+1):
        for j in range(1, m+1):
            if s1[j-1] == s2[i-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[n][m]
LCS("YXQYXP", "XYXZPQ")
```

Out[ ]: 4

o)

Tìm tập con của tập đã cho $a[1...n]$ gồm n số nguyên dương sao cho tổng của tập con này bằng với k

Ý tưởng:

- `dp[i][j]` sẽ cho biết liệu có thể đạt được tổng là `i` bằng cách sử dụng `j` phần tử đầu tiên của mảng hay không.

Công thức quy hoạch động tổng quát:

$$dp[i][j] = dp[i][j-1] \vee dp[i - a[j-1]][j-1]$$

- Nếu không chọn phần tử thứ j, tổng `i` vẫn có thể đạt được từ `j-1` phần tử.
- Nếu chọn phần tử thứ j, tổng `i` có thể đạt được nếu tồn tại một tổng $i - a[j-1]$ từ `j-1` phần tử đầu tiên.

Khởi tạo: $dp[0][j] = 1, \forall j$

Phân tích độ phức tạp:

- Thời gian: $O(k \times n)$, với k là tổng cần kiểm tra và n là số phần tử trong mảng.
- Không gian: $O(k \times n)$ cho mảng `dp`.

In [ ]:
```
def subsetSum(a, k):
    n = len(a)
    dp = [[0]*(n+1) for _ in range(k+1)]
    for i in range(n+1):
        dp[0][i] = 1
    for i in range(1, k+1):
        for j in range(1, n+1):
            dp[i][j] = dp[i][j-1]
            if i >= a[j-1]:
                dp[i][j] = dp[i][j] or dp[i-a[j-1]][j-1]
    return dp[k][n]
subsetSum([3,5,6,7],16)
```

Out[ ]: 1

Mở rộng 1 Cho tập A = {a1, a2, ..., an} gồm n số nguyên dương. Tìm cách chia tập này thành hai tập con A1 và A2 sao cho:

1. $A1 \cap A2 = \emptyset$, $A1 \cup A2 = A$
2. Tổng của các phần tử của hai tập bằng nhau.

Ý tưởng:

- Tìm xem liệu có thể có một tập con của `A` có tổng bằng một nửa tổng của tất cả các phần tử trong `A` hay không. Nếu có, thì tập con còn lại sẽ có tổng bằng nửa còn lại, tức là chúng ta có thể chia `A` thành hai tập con với tổng bằng nhau.

Công thức quy hoạch động tổng quát

- Với mỗi phần tử `a[i]` trong `A` và duyệt ngược từ `subset_sum` (là `S/2`) về `a[i]`:

$$dp[j] = dp[j] \text{ or } dp[j - a[i]]$$

- `dp[0] = True` (tổng bằng `0` luôn đạt được).

Phân tích độ phức tạp của thuật toán

- **Độ phức tạp thời gian**: `O(n * subset_sum)`, trong đó `n` là số phần tử trong `A` và `subset_sum` là `S/2`.
- **Độ phức tạp không gian**: `O(subset_sum)` do chỉ sử dụng mảng `dp[]` có kích thước `subset_sum + 1`.

In [ ]:
```python
def canPartition(a):
    S = sum(a)

    if S % 2 != 0:
        return False

    subset_sum = S // 2
    n = len(a)

    dp = [False] * (subset_sum + 1)
    dp[0] = True

    for i in range(n):
        for j in range(subset_sum, a[i] - 1, -1):
            if dp[j - a[i]]:
                dp[j] = True
    return dp[subset_sum]

A = [1, 5, 11, 5]
canPartition(A)
```

Out[ ]:
```
True
```

Mở rộng 2 Cho tập A = {a1, a2, ..., an} gồm n số nguyên dương. Tìm cách chia tập này thành hai tập con A1 và A2 thỏa những điều kiện sau:

1. $A1 \cap A2 = \emptyset$, $A1 \cup A2 = A$
2. Nếu gọi S_A là tổng của các phần tử của tập A thì $|S_{A1} - S_{A2}|$ là nhỏ nhất.

Ý tưởng:

- Tìm cách xác định tổng lớn nhất có thể của một tập con mà không vượt quá một nửa tổng của cả tập hợp. Sau đó, chúng ta sẽ tính toán sự chênh lệch giữa tổng của hai tập con dựa trên tổng này.

Công thức quy hoạch động tổng quát

1. **Khởi tạo**:
   - Gọi `S` là tổng của các phần tử trong tập hợp `A`.
   - Đặt `subset_sum = S // 2`.
   - Khởi tạo mảng `dp[]` với `dp[0] = True` và tất cả các giá trị khác ban đầu là `False`.
2. **Cập nhật**:

- Với mỗi phần tử `a[i]` trong `A`, ta cập nhật mảng `dp[]` như sau:

$$\text{dp[j]} = \text{dp[j] or dp[j - a[i]]}$$

- Cập nhật giá trị `max_subset_sum`:

$$max\_subset\_sum = \max(max\_subset\_sum, j)$$

khi `dp[j]` bằng `True`.

3. **Kết quả**:

- Sự chênh lệch nhỏ nhất giữa hai tổng là `S - 2 * max_subset_sum`.

Phân tích độ phức tạp của thuật toán

- **Độ phức tạp thời gian**: `O(n * subset_sum)` với `n` là số phần tử trong tập hợp `A` và `subset_sum` là `S // 2`.
- **Độ phức tạp không gian**: `O(subset_sum)` do chỉ sử dụng mảng `dp[]` với kích thước `subset_sum + 1`.

In [ ]:
```python
def minDifference(a):
    S = sum(a)
    n = len(a)

    subset_sum = S // 2
    dp = [False] * (subset_sum + 1)
    dp[0] = True

    max_subset_sum = 0

    for i in range(n):
        for j in range(subset_sum, a[i] - 1, -1):
            if dp[j - a[i]]:
                dp[j] = True
                max_subset_sum = max(max_subset_sum, j)

    return S - 2 * max_subset_sum

A = [8, 6, 11, 5]
minDifference(A)
```

Out[ ]:
```
2
```