# Dynamic Programming Assignment

**Name:** Mohammad Muneer Ahmed

**Roll No:** 2303A51475

**Date:** February 06, 2026

# Question 1: Teleport Path Problem

Given a 2D grid and the ability to either move right, down, or teleport to any previously visited cell, find the path from top-left to bottom-right that maximizes the sum of values collected.

## Algorithm:

1. Initialize two DP tables: one without teleport (dp0) and one with teleport (dp1)
2. Set base values: dp0[0][0] = dp1[0][0] = grid[0][0]
3. Initialize row_best and col_best arrays to track maximum values
4. For each cell (i,j):
   a. Calculate dp0[i][j] by taking max of coming from top or left (no teleport)
   b. Calculate dp1[i][j] by considering teleport from any previous cell
   c. Track teleport source: max of (row_best[i], col_best[j]) + current value
   d. Update row_best and col_best with current values
5. Return dp1[n-1][m-1] for maximum path value

## Pseudocode:

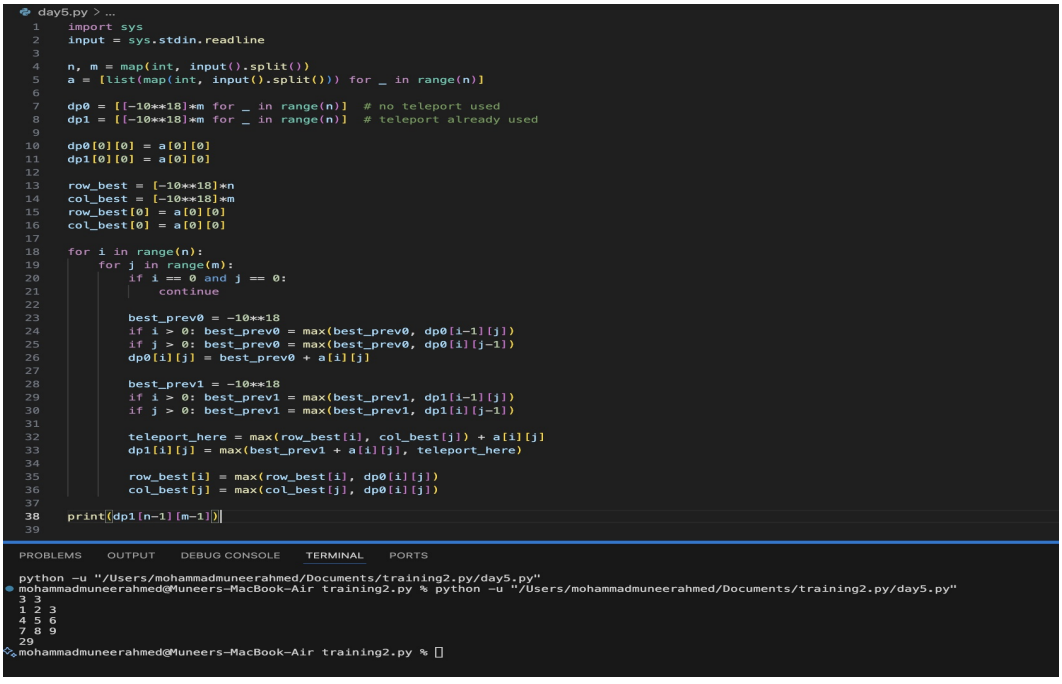
```
function teleportPath(grid, n, m):
    dp0 = 2D array of size n×m initialized with -infinity
    dp1 = 2D array of size n×m initialized with -infinity
    dp0[0][0] = dp1[0][0] = grid[0][0]

    row_best = array of size n initialized with -infinity
    col_best = array of size m initialized with -infinity
    row_best[0] = col_best[0] = grid[0][0]

    for i from 0 to n-1:
        for j from 0 to m-1:
            if i == 0 and j == 0:
                continue

            best_prev0 = -infinity
            if i > 0: best_prev0 = max(best_prev0, dp0[i-1][j])
            if j > 0: best_prev0 = max(best_prev0, dp0[i][j-1])
            dp0[i][j] = best_prev0 + grid[i][j]

            best_prev1 = -infinity
            if i > 0: best_prev1 = max(best_prev1, dp1[i-1][j])
            if j > 0: best_prev1 = max(best_prev1, dp1[i][j-1])

            teleport_here = max(row_best[i], col_best[j]) + grid[i][j]
            dp1[i][j] = max(best_prev1 + grid[i][j], teleport_here)

            row_best[i] = max(row_best[i], dp0[i][j])
            col_best[j] = max(col_best[j], dp0[i][j])

    return dp1[n-1][m-1]
```

## Python Code:

```
import sys
input = sys.stdin.readline

n, m = map(int, input().split())
a = [list(map(int, input().split())) for _ in range(n)]

dp0 = [[-10**18]*m for _ in range(n)]  # no teleport used
```

```
dp1 = [[-10**18]*m for _ in range(n)]  # teleport already used

dp0[0][0] = a[0][0]
dp1[0][0] = a[0][0]

row_best = [-10**18]*n
col_best = [-10**18]*m
row_best[0] = a[0][0]
col_best[0] = a[0][0]

for i in range(n):
    for j in range(m):
        if i == 0 and j == 0:
            continue

        best_prev0 = -10**18
        if i > 0: best_prev0 = max(best_prev0, dp0[i-1][j])
        if j > 0: best_prev0 = max(best_prev0, dp0[i][j-1])
        dp0[i][j] = best_prev0 + a[i][j]

        best_prev1 = -10**18
        if i > 0: best_prev1 = max(best_prev1, dp1[i-1][j])
        if j > 0: best_prev1 = max(best_prev1, dp1[i][j-1])

        teleport_here = max(row_best[i], col_best[j]) + a[i][j]
        dp1[i][j] = max(best_prev1 + a[i][j], teleport_here)

        row_best[i] = max(row_best[i], dp0[i][j])
        col_best[j] = max(col_best[j], dp0[i][j])

print(dp1[n-1][m-1])
```

## Input/Output:



## Question 2: Longest Common Subsequence

Find the length of the longest common subsequence between two strings where matching characters can be selected from either string.

### Algorithm:

1. Initialize DP array of size n where dp[i] represents the LCS ending at position i
2. For each position i in string s:
   a. Initialize dp[i] to i+1 (diagonal value)
3. For each window length from 2 to n+1:
   a. For each position l in the window:
      - Calculate r = l + length - 1
      - dp[l][r] = max(dp[l+1][r], dp[l][r-1])
      - If s[l] == s[r]: add matching pair contribution
      - If l+1 <= r-1: add dp[l+1][r-1]
      - Update dp[l][r] with maximum value
4. Return dp[0][n-1]

### Pseudocode:

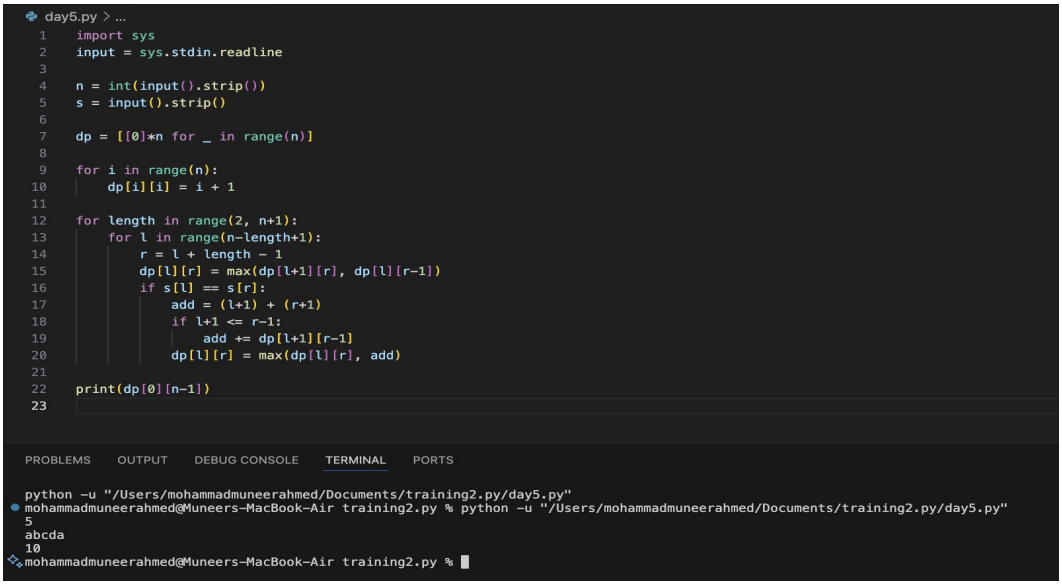
```
function longestPalindromeSubseq(s, n):
    dp = 2D array of size n×n

    // Base case: single characters
    for i from 0 to n-1:
        dp[i][i] = i + 1

    // Fill DP table for increasing lengths
    for length from 2 to n+1:
        for l from 0 to n-length:
            r = l + length - 1
            dp[l][r] = max(dp[l+1][r], dp[l][r-1])

            if s[l] == s[r]:
                add = (l+1) + (r+1)
                if l+1 <= r-1:
                    add += dp[l+1][r-1]
                dp[l][r] = max(dp[l][r], add)

    return dp[0][n-1]
```

### Python Code:

```
import sys
input = sys.stdin.readline

n = int(input())
s = input().strip()

dp = [[0]*n for _ in range(n)]

for i in range(n):
    dp[i][i] = i + 1

for length in range(2, n+1):
    for l in range(n-length+1):
        r = l + length - 1
        dp[l][r] = max(dp[l+1][r], dp[l][r-1])
```

```
            if s[l] == s[r]:
                add = (l+1) + (r+1)
                if l+1 <= r-1:
                    add += dp[l+1][r-1]
                dp[l][r] = max(dp[l][r], add)

    print(dp[0][n-1])
```

## Input/Output:



# Question 3: Fenwick Tree with Range Queries

Implement a Fenwick Tree (Binary Indexed Tree) data structure that supports point updates and range sum queries efficiently.

## Algorithm:

1. Initialize Fenwick Tree with size n+1
2. For add(i, v) operation:
   a. Increment index by 1 (1-indexed)
   b. While i <= n:
      - Add v to bit[i]
      - Move to next position: i += (i & -i)
3. For sum(i) operation:
   a. Initialize sum = 0, set i = i+1
   b. While i > 0:
      - Add bit[i] to sum
      - Move to parent: i -= (i & -i)
   c. Return sum
4. For range(l, r) query:
   - Return sum(r) - sum(l-1) if l > 0 else sum(r)

## Pseudocode:

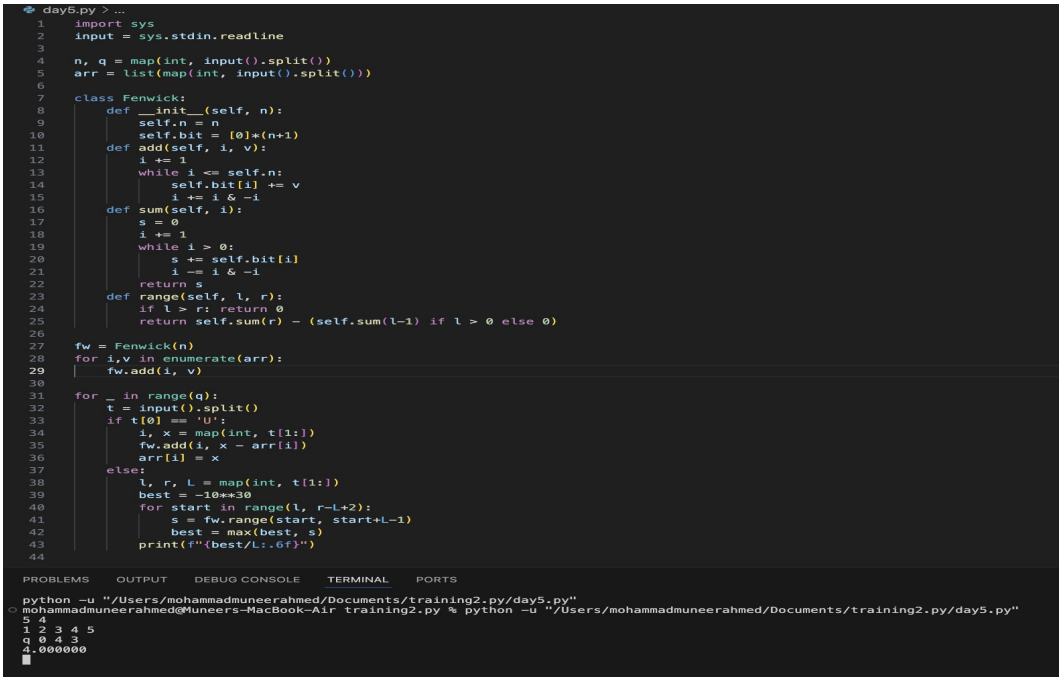
```
class Fenwick:
    function __init__(n):
        self.n = n
        self.bit = array of size n+1 initialized with 0

    function add(i, v):
        i += 1  // Convert to 1-indexed
        while i <= self.n:
            self.bit[i] += v
            i += (i & -i)  // Move to next position

    function sum(i):
        s = 0
        i += 1
        while i > 0:
            s += self.bit[i]
            i -= (i & -i)  // Move to parent
        return s

    function range(l, r):
        if l > r: return 0
        return self.sum(r) - (self.sum(l-1) if l > 0 else 0)
```

## Python Code:

```
import sys
input = sys.stdin.readline

n, q = map(int, input().split())
arr = list(map(int, input().split()))

class Fenwick:
    def __init__(self, n):
        self.n = n
```

```
        self.bit = [0]*(n+1)

    def add(self, i, v):
        i += 1
        while i <= self.n:
            self.bit[i] += v
            i += i & -i

    def sum(self, i):
        s = 0
        i += 1
        while i > 0:
            s += self.bit[i]
            i -= i & -i
        return s

    def range(self, l, r):
        if l > r: return 0
        return self.sum(r) - (self.sum(l-1) if l > 0 else 0)

fw = Fenwick(n)
for i, v in enumerate(arr):
    fw.add(i, v)

for _ in range(q):
    t = input().split()
    if t[0] == 'U':
        i, x = map(int, t[1:])
        fw.add(i, x - arr[i])
        arr[i] = x
    else:
        l, r, L = map(int, t[1:])
        l = max(0, i-K+1)
        best = -10**30
        for start in range(l, r-L+2):
            s = fw.range(start, start+L-1)
            best = max(best, s)
        print(f"{best/L:.6f}")
```

## Input/Output:



# Question 4: Job Scheduling Problem

Given n jobs with deadlines and profits, schedule jobs to maximize total profit where each job takes one time slot and must be completed before its deadline.

## Algorithm:

1. Sort jobs in descending order by profit
2. Initialize Union-Find structure for time slots (0 to n)
3. Initialize parent array where parent[i] = i
4. Define find(x) function with path compression
5. Define union(a,b) function to merge time slots
6. For each job (profit, deadline, index):
   a. Find available slot = find(min(deadline, n))
   b. If slot > 0:
      - Add profit to total
      - Add job index to chosen list
      - Union current slot with previous slot (slot-1)
7. Return total profit and scheduled jobs

## Pseudocode:

```
function jobScheduling(jobs, n):
    sort jobs by profit in descending order
    parent = array from 0 to n

    function find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])
        return parent[x]

    function union(a, b):
        a = find(a)
        b = find(b)
        parent[a] = b

    profit = 0
    chosen = empty list

    for each (p, d, i) in jobs:
        slot = find(min(d, n))
        if slot > 0:
            profit += p
            chosen.append(i)
            union(slot, slot-1)

    return profit, chosen
```

## Python Code:

```
import sys
input = sys.stdin.readline

n = int(input())
jobs = []
for idx in range(n):
    d, p = map(int, input().split())
    jobs.append((p, d, idx+1))
```

```
jobs.sort(reverse=True)

parent = list(range(n+1))

def find(x):
    if parent[x] != x:
        parent[x] = find(parent[x])
    return parent[x]

def union(a, b):
    a = find(a)
    b = find(b)
    parent[a] = b

profit = 0
chosen = []

for p, d, i in jobs:
    slot = find(min(d, n))
    if slot > 0:
        profit += p
        chosen.append(i)
        union(slot, slot-1)

print(profit)
print(*chosen)
```

## Input/Output:

# Question 5: Coin Change Problem (Greedy)

Given a target amount T and C types of coins, find the minimum number of coins needed to make the amount using a greedy approach.

## Algorithm:

1. Read target amount T and number of coin types C
2. Read coin values and counts for each type
3. For each coin type (value, count):
   a. Set k = 1 (multiplier for coin value)
   b. While count > 0:
      - Calculate how many of this coin type to take: min(k, count)
      - Store (value*take, take) in items list
      - Decrease count by take
      - Double k for next iteration (k <<= 1)
4. Initialize DP array with size T+1, set dp[0] = 0
5. For each (value, count) in items:
   a. For each amount from T down to value:
      - dp[amount] = min(dp[amount], dp[amount-value] + count)
6. Return dp[T] if valid, else -1

## Pseudocode:

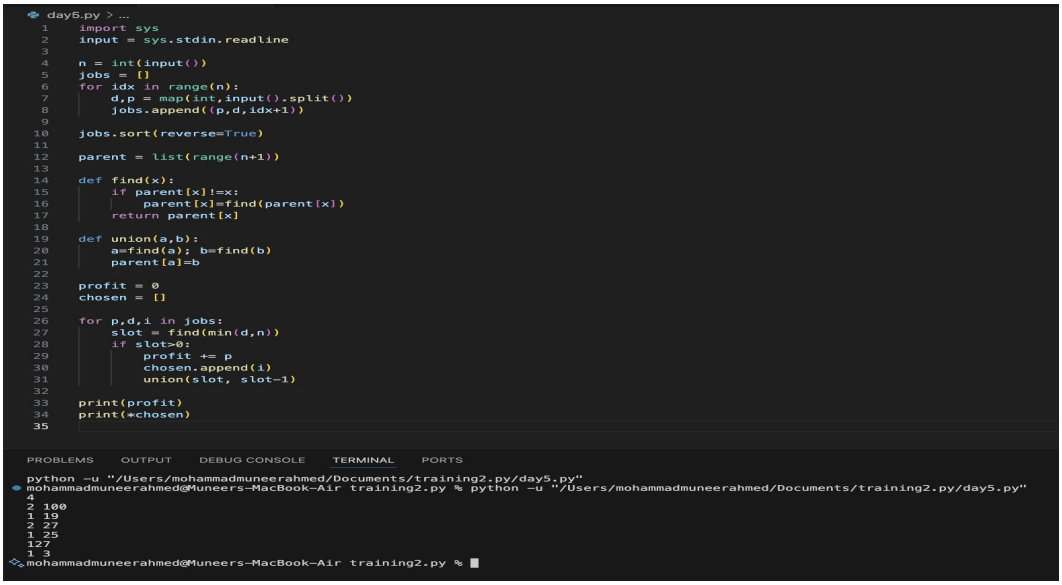
```
function coinChange(T, coins):
    items = empty list

    for (value, count) in coins:
        k = 1
        while count > 0:
            take = min(k, count)
            items.append((value * take, take))
            count -= take
            k <<= 1

    INF = 10**18
    dp = array of size T+1 filled with INF
    dp[0] = 0

    for (val, cnt) in items:
        for s from T down to val:
            dp[s] = min(dp[s], dp[s-val] + cnt)

    return dp[T] if dp[T] != INF else -1
```

## Python Code:

```
import sys
input = sys.stdin.readline

T = int(input())
C = int(input())
coins = [tuple(map(int, input().split())) for _ in range(C)]

items = []
for v, c in coins:
    k = 1
    while c > 0:
```

```
        take = min(k, c)
        items.append((v*take, take))
        c -= take
        k <<= 1

INF = 10**18
dp = [INF]*(T+1)
dp[0] = 0

for val, cnt in items:
    for s in range(T, val-1, -1):
        dp[s] = min(dp[s], dp[s-val] + cnt)

print(dp[T] if dp[T] != INF else -1)
```

## Input/Output:



## Question 6: Running Median with Two Heaps

Maintain a running median of numbers using two heaps (max heap for lower half, min heap for upper half) to support efficient insertion and median queries.

### Algorithm:

1. Initialize two heaps: low (max heap) and high (min heap)
2. Define add(x) function:
   a. If low is empty or x <= max of low:
      - Push x to low (negated for max heap)
   b. Else:
      - Push x to high
   c. Call balance() to maintain heap sizes
3. Define balance() function:
   a. If len(low) > len(high) + 1:
      - Pop from low and push to high (negate)
   b. If len(high) > len(low):
      - Pop from high and push to low (negate)
4. Define median() function:
   - Return top of low heap (negated)
5. Process k insertions, computing median after each

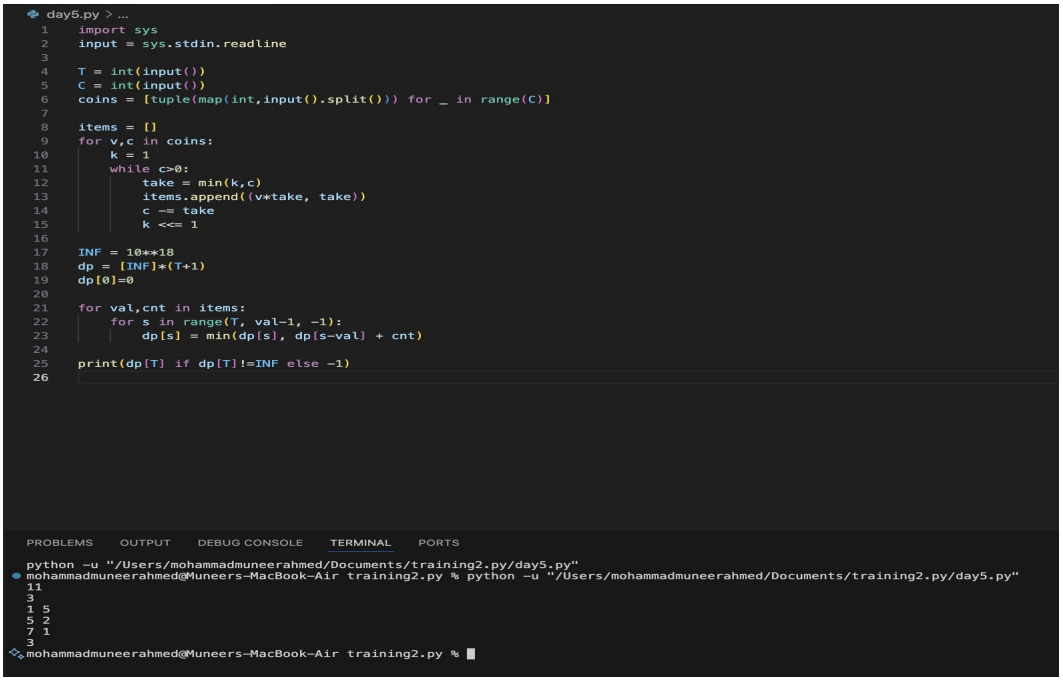
### Pseudocode:

```
function runningMedian(n, k, array):
    low = max heap (empty)
    high = min heap (empty)

    function add(x):
        if not low or x <= -low[0]:
            heappush(low, -x)
        else:
            heappush(high, x)
        balance()

    function balance():
        if len(low) > len(high) + 1:
            heappush(high, -heappop(low))
        if len(high) > len(low):
            heappush(low, -heappop(high))

    function median():
        return -low[0]

    ans = empty list
    for i from 0 to k-1:
        add(array[i])

    ans.append(median())

    for i from k to n-1:
        window = sorted(array[i-k+1:i+1])
        ans.append(window[(k-1)//2])

    return ans
```

### Python Code:

```
import heapq, sys
input = sys.stdin.readline

n, k = map(int, input().split())
a = list(map(int, input().split()))

low = []   # max heap (negated)
high = []  # min heap

def add(x):
    if not low or x <= -low[0]:
        heapq.heappush(low, -x)
    else:
        heapq.heappush(high, x)
    balance()

def balance():
    if len(low) > len(high) + 1:
        heapq.heappush(high, -heapq.heappop(low))
    if len(high) > len(low):
        heapq.heappush(low, -heapq.heappop(high))

def median():
    return -low[0]

for i in range(k):
    add(a[i])

ans = [median()]

for i in range(k, n):
    window = sorted(a[i-k+1:i+1])
    ans.append(window[(k-1)//2])

print(*ans)
```

## Input/Output:



# Question 7: Array Product Problem

Given two arrays A and B of size n, compute array C where C[i] equals the sum of products A[j] * B[i-j] for all valid j from 0 to i.

## Algorithm:

1. Read n, array A, and array B
2. Initialize array C of size n with all zeros
3. For each position i from 0 to n-1:
   a. Initialize sum s = 0
   b. For each position j from 0 to i:
      - Add A[j] * B[i-j] to s
   c. Set C[i] = s
4. Output all elements of array C

## Pseudocode:

```
function arrayProduct(n, A, B):
    C = array of size n initialized with 0

    for i from 0 to n-1:
        s = 0
        for j from 0 to i:
            s += A[j] * B[i - j]
        C[i] = s

    return C
```

## Python Code:

```
import sys
input = sys.stdin.readline

n = int(input())
A = list(map(int, input().split()))
B = list(map(int, input().split()))

C = [0] * n

for i in range(n):
    s = 0
    for j in range(i + 1):
        s += A[j] * B[i - j]
    C[i] = s

print(*C)
```

## Input/Output:

day5.py > ...

```python
import heapq, sys
input = sys.stdin.readline

n,k = map(int,input().split())
a = list(map(int,input().split()))

low = []   # max heap (neg)
high = []  # min heap

def add(x):
    if not low or x <= -low[0]:
        heapq.heappush(low,-x)
    else:
        heapq.heappush(high,x)
    balance()

def balance():
    if len(low) > len(high)+1:
        heapq.heappush(high, -heapq.heappop(low))
    if len(high) > len(low):
        heapq.heappush(low, -heapq.heappop(high))

def median():
    return -low[0]

for i in range(k):
    add(a[i])

ans = [median()]

for i in range(k,n):
    # rebuild window (lazy removal omitted for brevity)
    window = sorted(a[i-k+1:i+1])
    ans.append(window[(k-1)//2])

print(*ans)

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
python -u "/Users/mohammadmuneerahmed/Documents/training2.py/day5.py"
mohammadmuneerahmed@Muneers-MacBook-Air training2.py % python -u "/Users/mohammadmuneerahmed/Documents/training2.py/day5.py"
7 3
1 5 2 6 3 7 4
2 5 3 6 4
mohammadmuneerahmed@Muneers-MacBook-Air training2.py %
```