Question 1 — normalize_runs(original, token)

Idea

Scan original left→right.
Whenever we see token starting at position i, we:

1. append **one** token to the output
2. skip **all consecutive** repeats of token
   Otherwise, copy one character and advance by 1.

No replace(), no regex.

Code

```
def normalize_runs(original, token):
  if token == "":
    # Not expected in typical exams, but avoids infinite loops.
    return original
  n = len(original)
  m = len(token)
  out = []
  i = 0
  found = False
  while i < n:
    # Check if token matches starting at i
    if i + m <= n and original[i:i+m] == token:
      found = True
      out.append(token)
      i += m
      # Skip any further consecutive tokens
      while i + m <= n and original[i:i+m] == token:
        i += m
    else:
      out.append(original[i])
      i += 1
```

result = "".join(out)

return result if found else original

Controlled execution (trace)

Example: normalize_runs("ab--cd---ef-", "-")

- token="-" (m=1)
- Start: i=0, out=[]

1. i=0 → "a" not token → out=["a"], i=1
2. i=1 → "b" not token → out=["a","b"], i=2
3. i=2 → "-" matches → append "-" → out=["a","b","-"], i=3
    - inner skip: i=3 is "-" → skip → i=4
4. i=4 → "c" → out=["a","b","-","c"], i=5
5. i=5 → "d" → out=["a","b","-","c","d"], i=6
6. i=6 → "-" matches → append "-" → out=["a","b","-","c","d","-"], i=7
    - skip: i=7 is "-" → i=8
    - skip: i=8 is "-" → i=9
7. i=9 → "e" → out=...,"e", i=10
8. i=10 → "f" → out=...,"f", i=11
9. i=11 → "-" matches → append "-" → out=...,"-", i=12 end

Final output: "ab-cd-ef-"

Question 2 — Warmer temperature waiting days

We want, for each day i, the distance to the **next** day j>i with temps[j] > temps[i], else 0.

Part A — Python lists (efficient O(n) using a stack)

This is the classic **monotonic decreasing stack** of indices.

Code

def daily_warmer_days_list(temps):

n = len(temps)

```
    ans = [0] * n
    stack = []  # will store indices; temps[stack] is decreasing
    for i, t in enumerate(temps):
        while stack and t > temps[stack[-1]]:
            prev = stack.pop()
            ans[prev] = i - prev
        stack.append(i)
    return ans
```

Controlled execution (trace)

For temps = [73, 74, 75, 71, 69, 72, 76, 73]

We track (i, t), stack (indices), ans updates:

- Start: stack=[], ans=[0,0,0,0,0,0,0,0]

1. i=0,t=73: stack empty → push 0 → stack=[0]
2. i=1,t=74: 74>73 → pop 0 → ans[0]=1
   push 1 → stack=[1]
3. i=2,t=75: 75>74 → pop 1 → ans[1]=1
   push 2 → stack=[2]
4. i=3,t=71: 71 not >75 → push 3 → stack=[2,3]
5. i=4,t=69: 69 not >71 → push 4 → stack=[2,3,4]
6. i=5,t=72: 72>69 → pop 4 → ans[4]=1
   72>71 → pop 3 → ans[3]=2
   72 not >75 → push 5 → stack=[2,5]
7. i=6,t=76: 76>72 → pop 5 → ans[5]=1
   76>75 → pop 2 → ans[2]=4
   push 6 → stack=[6]
8. i=7,t=73: 73 not >76 → push 7 → stack=[6,7]

End: remaining indices [6,7] have no warmer day → keep 0.

Final: **[1, 1, 4, 2, 1, 1, 0, 0]**

Part B (14 pts) — NumPy "vectorized operations"

True *fully vectorized* "next greater element" isn't natural; a common exam-acceptable approach is an **O(n^2) vectorized mask** (fine for moderate n).

Code (vectorized mask + row-wise min)

```
import numpy as np

def daily_warmer_days_numpy(temps):
    T = np.asarray(temps)
    n = T.size
    idx = np.arange(n)
    # mask[i, j] = (j > i) and (T[j] > T[i])
    mask = (T[None, :] > T[:, None])
    mask = np.triu(mask, k=1)  # keep only j>i
    # Convert True positions to their column index; others to n (sentinel)
    candidates = np.where(mask, idx[None, :], n)
    next_idx = candidates.min(axis=1)  # first warmer day index if exists, else n
    wait = np.where(next_idx < n, next_idx - idx, 0)
    return wait.tolist()
```

Controlled execution (trace)

Using the same example:

- T = [73,74,75,71,69,72,76,73], idx=[0..7]
- For row i=0 (73): warmer days are at j=1,2,6 → min j=1 → wait=1
- Row i=2 (75): warmer day at j=6 → wait=4
- Row i=6 (76): none → sentinel n → wait=0
  Result: **[1,1,4,2,1,1,0,0]**

Question 3 — unique_max_subarray_sum(arr) (NumPy array)

Goal

Maximum sum of a **contiguous** subarray with **all unique values**.

A robust approach (works even if negatives exist)

We maintain:

- unique_left: left bound forced by duplicates (via last_seen)
- prefix sums P[k] = sum(arr[:k])
- for each end r, the best unique subarray ending at r is:
  [
  \max_{l \in [unique_left, r]} (P[r+1] - P[l]) = P[r+1] - \min(P[l])
  ]
  So we need the minimum prefix sum in a sliding index range → maintain a deque of candidate prefix indices with increasing prefix sums.

Code

```
import numpy as np

from collections import deque

def unique_max_subarray_sum(arr):
    arr = np.asarray(arr)
    n = arr.size
    last_seen = {}        # value -> last index
    unique_left = 0       # minimal l allowed due to duplicates
    P = 0                 # current prefix sum P[r+1]
    prefix = [0]          # store prefix sums for indexing
    dq = deque([0])       # indices into prefix[], increasing by prefix value
    best = -10**18
    for r in range(n):
        x = int(arr[r])
        # Update uniqueness constraint
        if x in last_seen and last_seen[x] >= unique_left:
```

```
        unique_left = last_seen[x] + 1
    last_seen[x] = r
    # Extend prefix sum
    P += x
    prefix.append(P)  # prefix[r+1]
    # Remove deque indices that are left of the allowed range
    while dq and dq[0] < unique_left:
        dq.popleft()
    # Best subarray ending at r uses minimal prefix in allowed range
    best = max(best, prefix[r+1] - prefix[dq[0]])
    # Add index (r+1) as a future l-candidate; keep deque increasing by prefix value
    while dq and prefix[dq[-1]] >= prefix[r+1]:
        dq.pop()
    dq.append(r+1)
  return best
```

Controlled execution (trace)

Example: arr = [4, 2, 4, 5, 6]

Track: r, x, unique_left, prefix[r+1], dq (as indices), best

- Start: prefix=[0], dq=[0], unique_left=0, best=-inf

1. r=0,x=4: not seen
   prefix=4, dq valid
   candidate = prefix[1]-prefix[0]=4-0=4 → best=4
   add idx=1 (prefix=4): dq becomes [0,1]
2. r=1,x=2: not seen
   prefix=6
   candidate = 6 - min(prefix in dq)=6-0=6 → best=6
   add idx=2 (prefix=6): dq [0,1,2]
3. r=2,x=4: seen at 0, and 0>=unique_left(0) → unique_left=1
   prefix=10

pop dq front indices <1 → dq from [0,1,2] to [1,2]
candidate = 10 - prefix[1]=10-4=6 → best still 6
add idx=3 (prefix=10): dq [1,2,3]

4. r=3,x=5: not seen
   prefix=15
   candidate = 15 - prefix[1]=15-4=11 → best=11
   add idx=4 → dq [1,2,3,4]
5. r=4,x=6: not seen
   prefix=21
   candidate = 21 - prefix[1]=21-4=17 → best=17

Return **17** (subarray [2,4,5,6]) ✅

Question 4 (8 pts) — Deep copy vs aliasing

Code given

```
import copy

a = [10, [20, 30], 40]

b = a

c = copy.deepcopy(a)

a[1][0] = 99

b[2] = 88

print(c)
```

Explanation

- b = a means **b and a reference the same list**.
- c = copy.deepcopy(a) creates a **fully independent copy**, including the inner list.

Edits:

- a[1][0] = 99 changes the inner list of a (and also b, since same object).
- b[2] = 88 changes a[2] too.

But **c remains unchanged** (deep copy made before modifications).

Controlled execution (trace)

After c = deepcopy(a):

- a = [10, [20, 30], 40]
- b = same object as a
- c = independent [10, [20, 30], 40]

After a[1][0]=99:

- a = [10, [99, 30], 40]
- b = [10, [99, 30], 40]
- c = [10, [20, 30], 40]

After b[2]=88:

- a = [10, [99, 30], 88]
- b = [10, [99, 30], 88]
- c = [10, [20, 30], 40]

Printed: **[10, [20, 30], 40]** → **Option B**.

Question 5 (10 pts) — longest_palindrome(words)

Idea

A word is a palindrome if word == word[::-1].
Scan the list, keep the longest palindrome found. If none, return "None".

Code

```
def longest_palindrome(words):
    best = None
    for w in words:
        if w == w[::-1]:
            if best is None or len(w) > len(best):
                best = w
    return best if best is not None else "None"
```

Controlled execution (trace)

Example: words = ["apple", "noon", "level", "banana"]

- Start: best=None

1. "apple" == "elppa"? no → best=None
2. "noon" == "noon"? yes → best=None → best="noon"
3. "level" == "level"? yes, len=5 > len("noon")=4 → best="level"
4. "banana" palindrome? no → best remains "level"

Return "level".