Exercise lesson

Presented by: Huu Phong and Tien Huy

Problem 1:

Football Tourament

Description

Input:

Single integer

n (2<=<10^5),

x_i, y_i

(1<= x_i, y_i <= 10^5)

Constraints:

x_i >< y_i

Output:

the number of games this team will play in home and away kits.

Example 1:

THE NUMBER OF PARTICIPANT TEAMS: N = 2

THE COLOR NUMBER OF 1'ST TEAM: X_1, Y_1 = 1, 2

THE COLOR NUMBER OF 2'ND TEAM: X_2, Y_2 = 2, 1

Output:

2 0

2 0

Example 2:

THE NUMBER OF PARTICIPANT TEAMS: N = 3

THE COLOR NUMBER OF 1'ST TEAM: X_1, Y_1 = 1, 2

THE COLOR NUMBER OF 2'ND TEAM: X_2, Y_2 = 2, 1

THE COLOR NUMBER OF 3'RD TEAM: X_3, Y_3 = 1,3

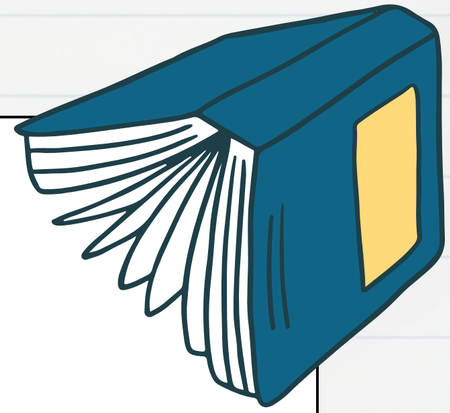
Output:

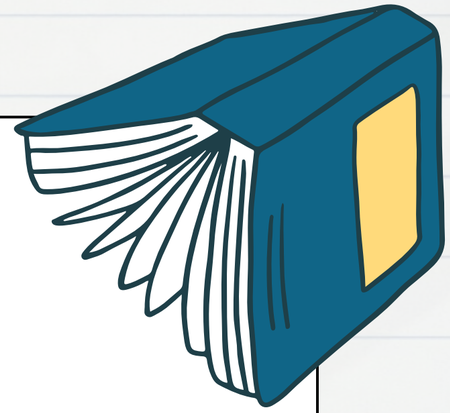
3 1

4 0

2 2

HOW WILL WE SOLVE IT?





- STEP 1: IN A GIVEN PROBLEM, FIND THE SUBPROBLEMS.
- STEP 2: DETERMINE WHAT THE SOLUTION WILL INCLUDE (E.X., LARGEST SUM, SHORTEST PATH).
- STEP 3: CREATE AN ITERATIVE PROCESS FOR GOING OVER ALL SUBPROBLEMS AND CREATING AN OPTIMUM SOLUTION.

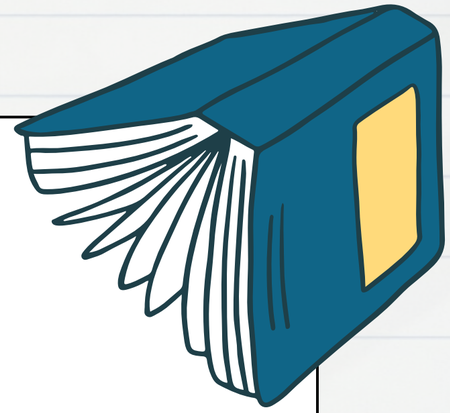
Simplifying the problem

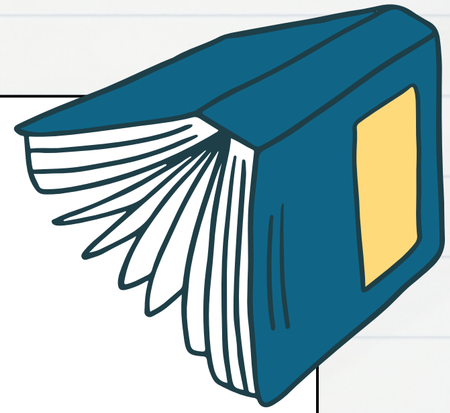
Assumption: (not true in reality)

The teams from the first leg will play at home, and the return leg will play away.

Code

```
N = INT(INPUT())
A = [0] * N
C = [0] * 100001
FOR I IN RANGE(N):
    S,  A[I] = MAP(INT,INPUT().SPLIT())
    C[S] += 1
FOR I IN RANGE(N):
    PRINT((N-1) + C[A[I]] , (N-1) - C[A[I]])
```





Complexity

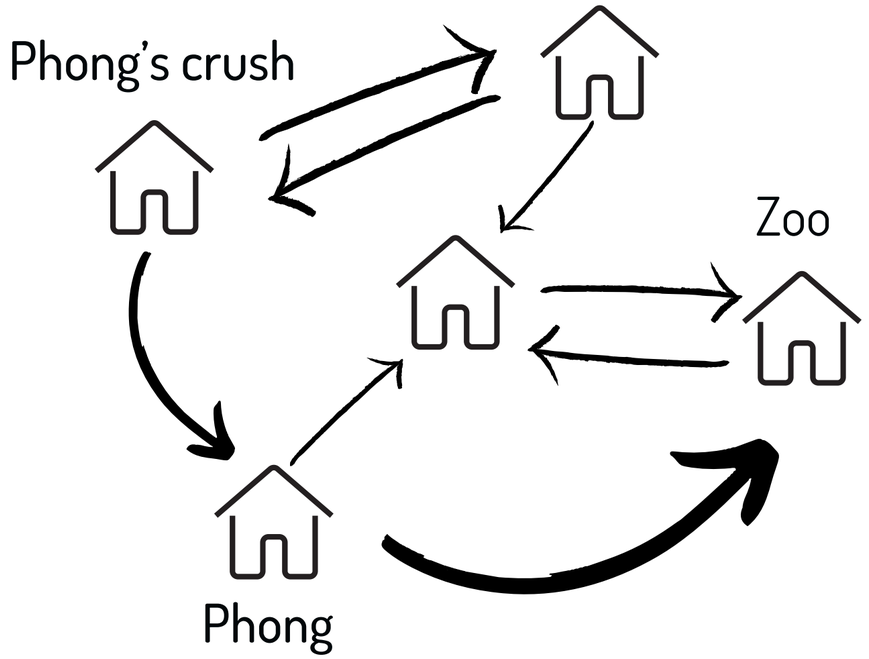
For loop

O(2n)

→

O(2n)

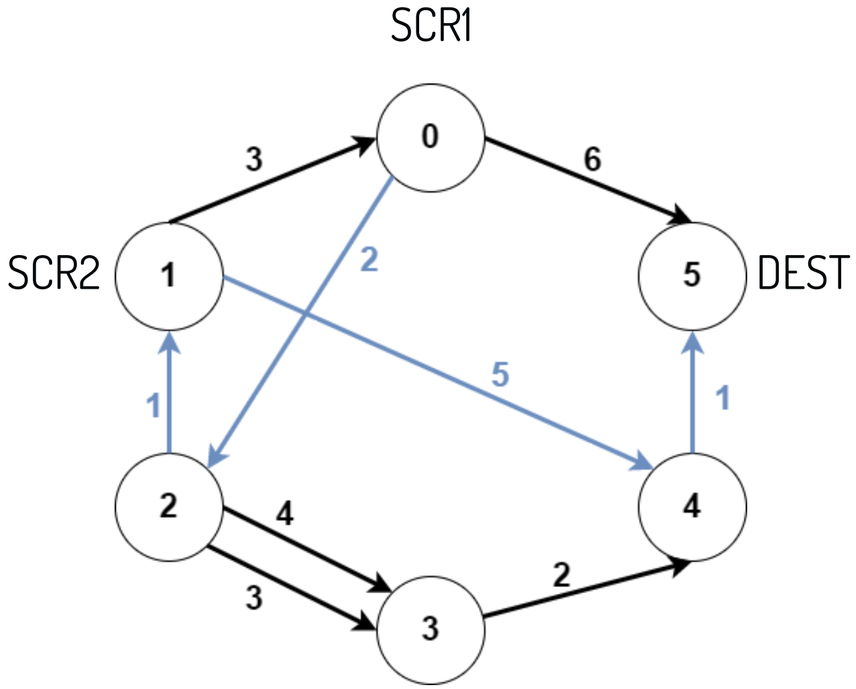
Problem 2



Problem 2

You are given **a weighted directed graph** and **distinct nodes** of the graph including **src1**, **src2** and **dest**. You have to find the minimum weight of a subgraph of the graph such that it is possible to reach **dest** from both **src1** and **src2** via a set of edges of this subgraph.

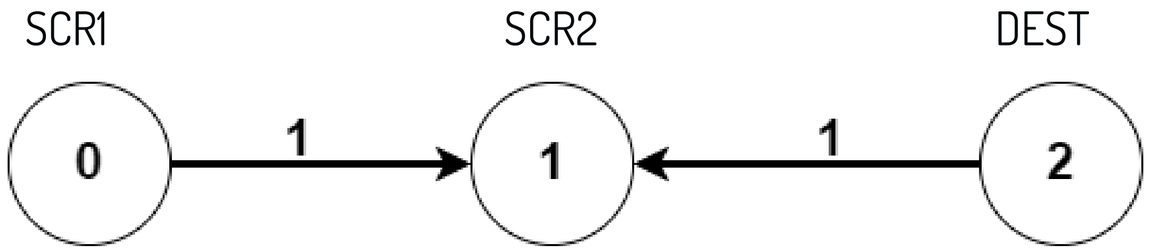
Examples



INPUT: N = 6, EDGES = [[0,2,2],[0,5,6],[1,0,3],[1,4,5],[2,1,1], [2,3,3],[2,3,4],[3,4,2],[4,5,1]], SRC1 = 0, SRC2 = 1, DEST = 5

OUTPUT: 9

Examples



INPUT: N = 3, EDGES = [[0,1,1],[2,1,1]],
SRC1 = 0, SRC2 = 1, DEST = 2
OUTPUT: -1

The minimum weight from **src1** to **src2** and opposite

The minimum weight from **dest** to **src1** and **src2**

The minimum weight of **subgraph**.

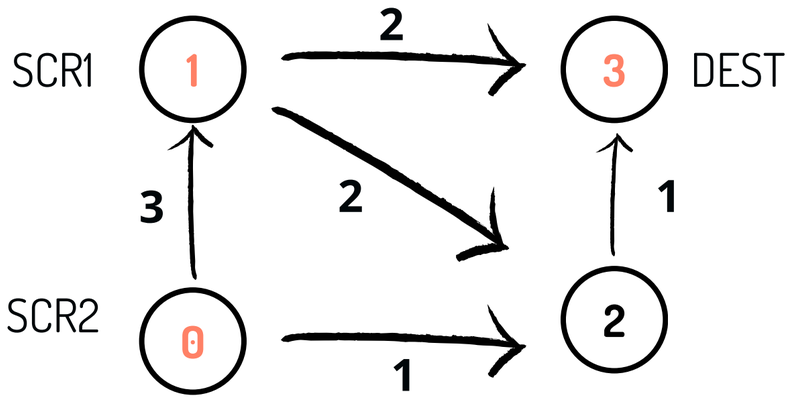
The minimum weight from **src1** to **src2** and opposite

The minimum weight from **dest** to **src1** and **src2**

The minimum weight of **subgraph**.

→ **THIS METHOD IS NOT OPTIMAL**

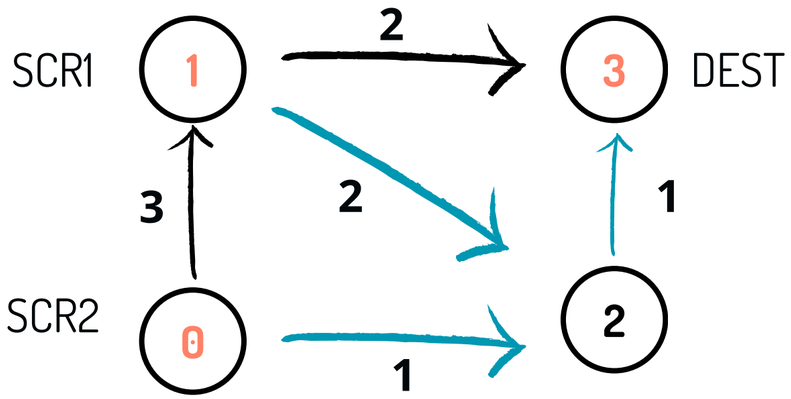
Examples



INPUT: N = 4, EDGES = [[0,2,2],[0,5,6],[1,0,3],[1,4,5],[2,1,1], [2,3,3],[2,3,4],[3,4,2],[4,5,1]], SRC1 = 0, SRC2 = 1, DEST = 3

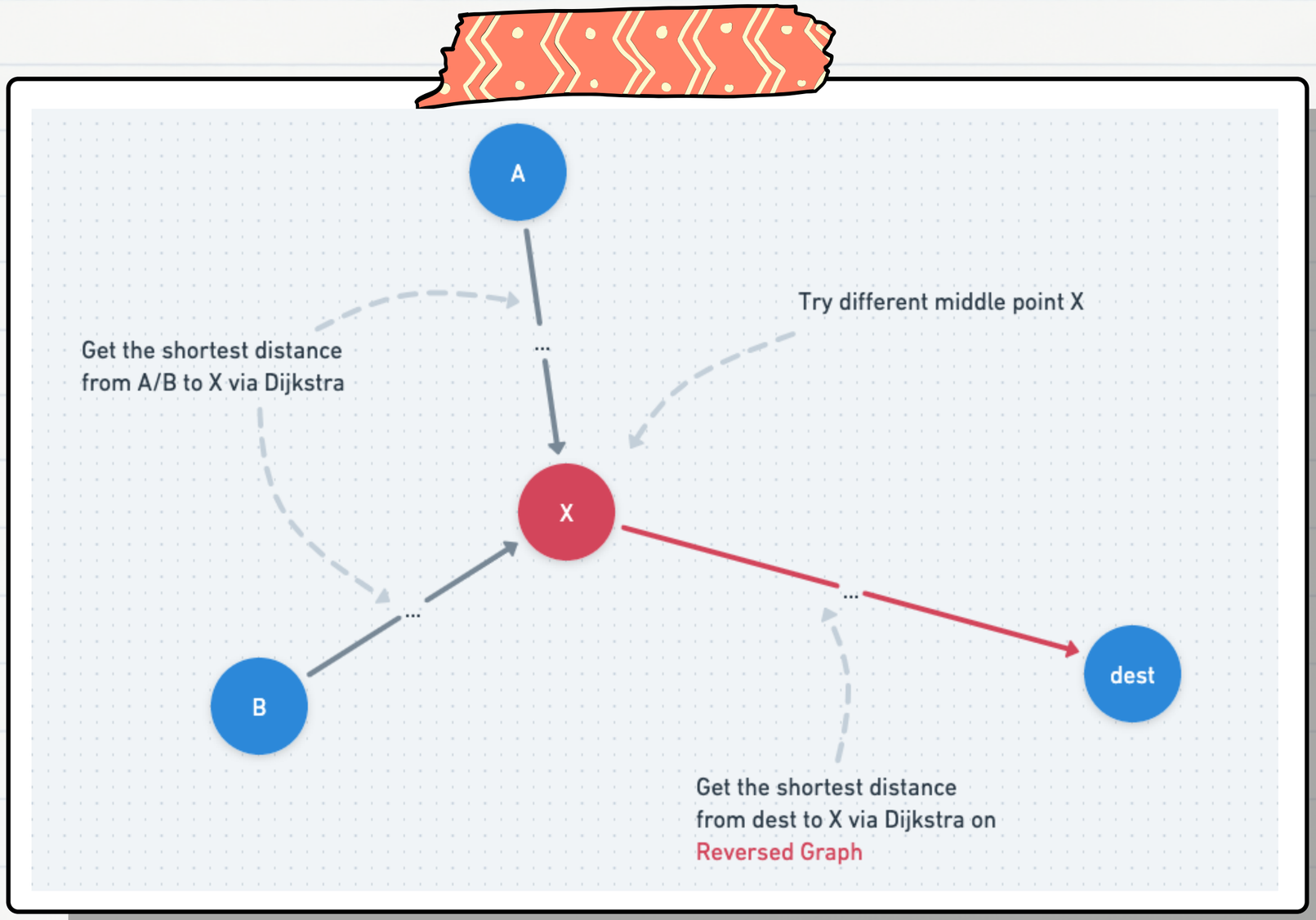
OUTPUT: 4

Examples



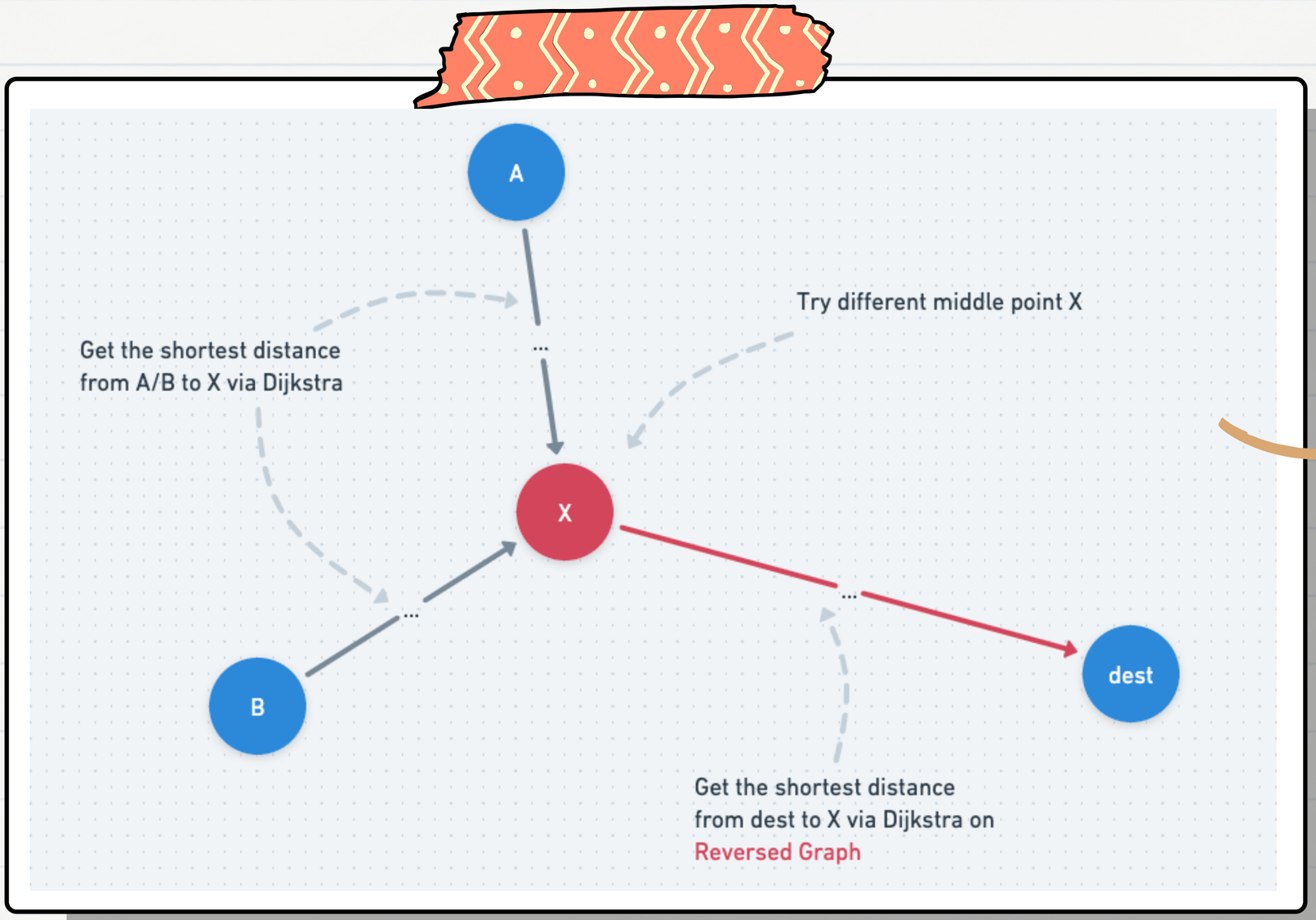
INPUT: N = 4, EDGES = [[0,1,3],[0,2,1],[1,2,2],[1,3,2],[2,3,1]]
SRC1 = 0, SRC2 = 1, DEST = 3
OUTPUT: 4

Approach


A
Try different middle point X
Get the shortest distance
from A/B to X via Dijkstra
...
X
...
B
dest
Get the shortest distance
from dest to X via Dijkstra on
Reversed Graph


OATS & MILK

Approach



Dijkstra 3 times

Dijkstra 3 times

SRC1

From s1 to x, for this we use Dijkstra

SRC2

From s2 to x, same.

DEST

From x to dest, for this we use Dijkstra on the reversed graph.

Code

Dijkstra

```python
def Dijkstra(graph, K):
    q, t = [(0, K)], {}
    while q:
        time, node = heappop(q)
        if node not in t:
            t[node] = time
            for v, w in graph[node]:
                heappush(q, (time + w, v))
    return [t.get(i, float("inf")) for i in range(n)]
```

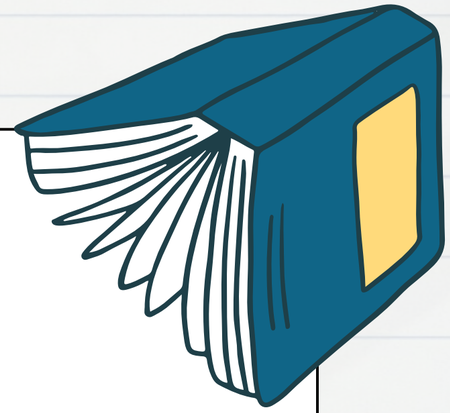
Code

```python
def minimumWeight(n, edges, s1, s2, dest):
        G1 = defaultdict(list)
        G2 = defaultdict(list)
        for a, b, w in edges:
            G1[a].append((b, w))
            G2[b].append((a, w))

        arr1 = Dijkstra(G1, s1)
        arr2 = Dijkstra(G1, s2)
        arr3 = Dijkstra(G2, dest)

        ans = float("inf")
        for i in range(n):
            ans = min(ans, arr1[i] + arr2[i] + arr3[i])

        return ans if ans != float("inf") else -1
```



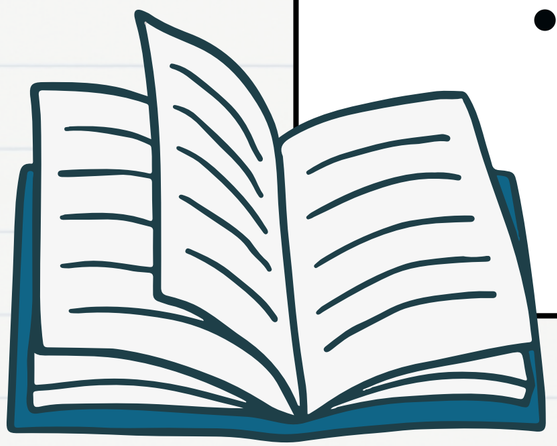
Complexity

Initialize Graph	Dijkstra	For loop
O(E)	O(ElogE)	O(n)

→ **O(2E + 3ElogE + n)**

Problem 3

- You are given an integer array. You can perform any number of operations, where each operation involves selecting a subarray of the array and replacing it with the sum of its elements.
- For example, if the given array is [1,3,5,6] and you select subarray [3,5] the array will convert to [1,8,6].
- Return the **maximum** length of a **non-decreasing** array that can be made after applying operations.
- Constraints:
  - 1 <= nums.length <= 10^5
  - 1 <= nums[i] <= 10^5



Examples

01

Input: nums = [5,2,2]

Output: 1

02

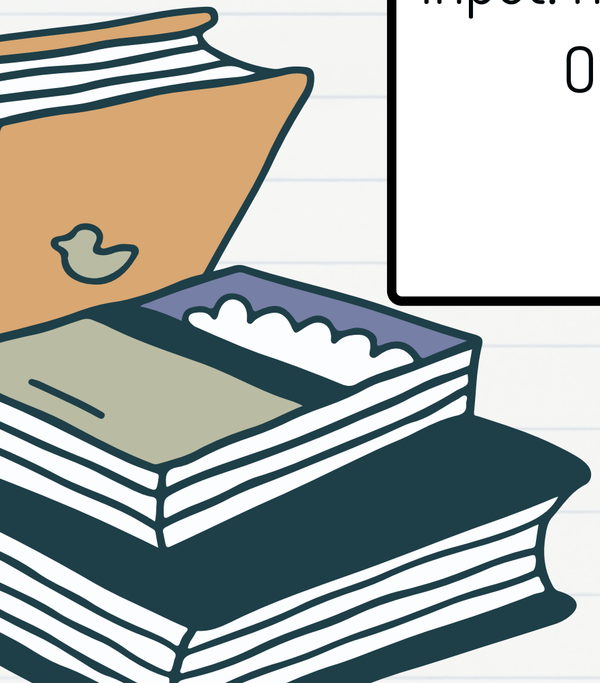
Input: nums = [1,2,3,4]

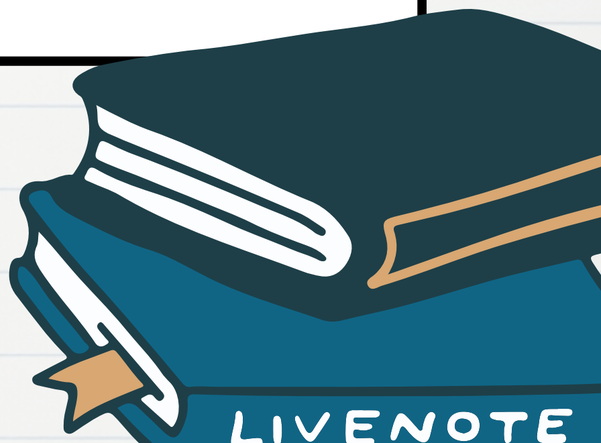
Output: 4

03

Input: nums = [4,3,2,6]

Output: 3





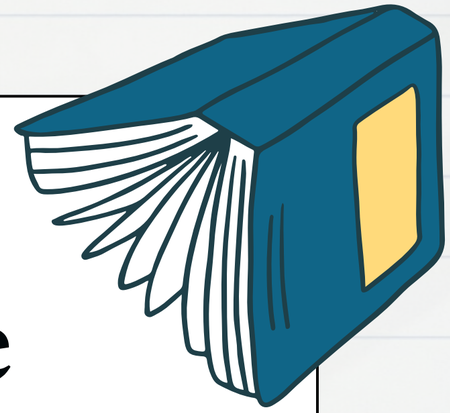
Analysis

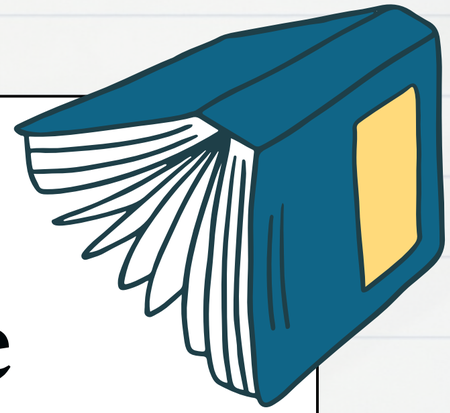
This problem essentially asks us to divide an array into subarrays so that the sum of each subarray is non-decreasing

divide array into subarrays:

4, 3, 2, 6, 1, 2, 1, 2, 8, 5, 3, 6, 11, 13, 4

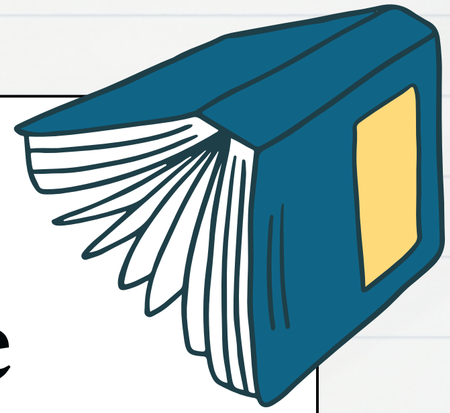
What is the algorithm that we should use?





What is the algorithm that we should use?

→ DYNAMIC PROGRAMMING



What is the algorithm that we should use?

→ DYNAMIC PROGRAMMING

Analysis

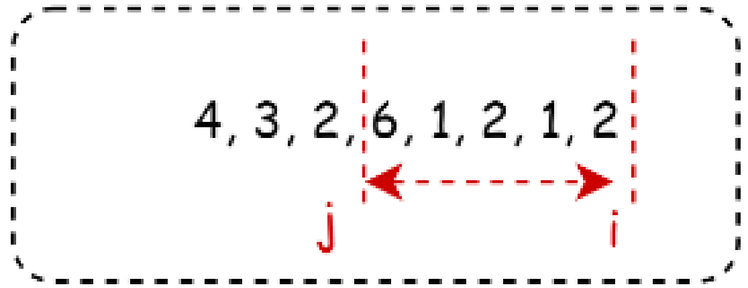
divide array into subarrays:

4, 3, 2, 6, 1, 2, 1, 2, 8, 5, 3, 6, 11, 13, 4

dp[i]: is the max possible number of subarrays for **i** first elements

→ *dp[i] <= dp[i+1]*

Analysis



So for each a[i], we want to find the biggest j that, a[j] + a[j + 1] .. + a[i] is big enough to make the applied array non decreasing, then dp[i + 1] = dp[j] + 1.

Brute-force

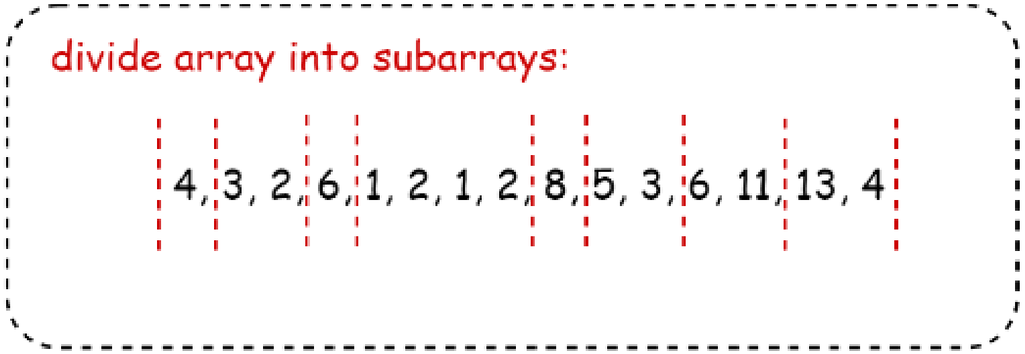
```python
def findMaximumLength(a: List[int]) -> int:
    n = len(A)
    last = [0] + [inf] * n
    prefix = [0]
    dp = [0] + [0] * n
    for i in range(n):
        prefix.append(a[i] + prefix[-1])
        j = i
        while last[j] > prefix[-1] - prefix[j]:
            j -= 1
        last[i + 1] = prefix[-1] - prefix[j]
        dp[i + 1] = dp[j] + 1
    return dp[-1]
```

Brute-force

```python
def findMaximumLength(a: List[int]) -> int:
    n = len(A)
    last = [0] + [inf] * n
    prefix = [0]
    dp = [0] + [0] * n
    for i in range(n):
        prefix.append(a[i] + prefix[-1])
        j = i
        while last[j] > prefix[-1] - prefix[j]:
            j -= 1
        last[i + 1] = prefix[-1] - prefix[j]
        dp[i + 1] = dp[j] + 1
    return dp[-1]
```

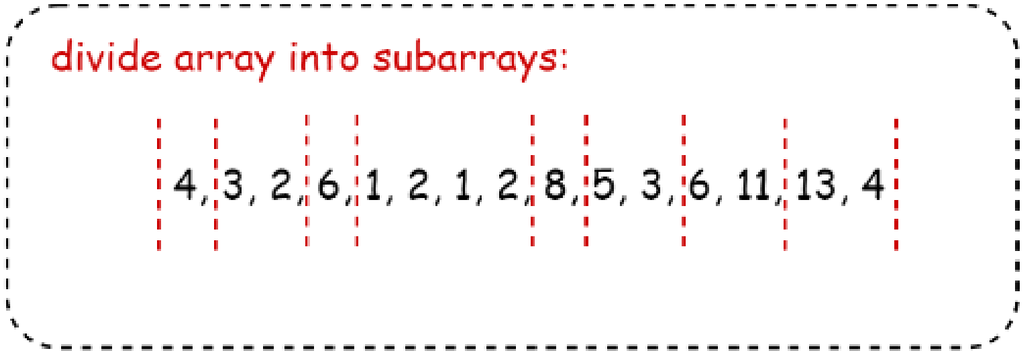
→ **Complexity: O(n^2)**

Binary Search



- The sum of last group:
  prefix[i] - prefix[j] = a[j] + ... + a[i - 1].
- Finding **k**:
  prefix[i] - prefix[j] <= prefix[k] - prefix[i]
- So that **acc[i] * 2 - acc[j] <= acc[k].**

Binary Search



- So we can apply binary search in acc array, to find the lower bound index of **prefix[i] * 2 - prefix[j].**
- And we assign **pre[k] = i**, which means to make the k first element non-decreasing,
- we need to at least to sum up **a[i] + a[i + 1] + .. a[k - 1]** to make it big enough.

Binary Search

```python
def findMaximumLength(A: List[int]) -> int:
    n = len(A)
    prefix = list(accumulate(A, initial = 0))
    pre = [0] * (n + 2)
    dp = [0] * (n + 1)
    j = 0
    for i,a in enumerate(A, 1):
        j = max(j, pre[i])
        dp[i] = dp[j] + 1
        k = bisect_left(prefix, prefix[i] * 2 - prefix[j])
        pre[k] = i
    return dp[n]
```

Binary Search

```python
def findMaximumLength(A: List[int]) -> int:
    n = len(A)
    prefix = list(accumulate(A, initial = 0))
    pre = [0] * (n + 2)
    dp = [0] * (n + 1)
    j = 0
    for i,a in enumerate(A, 1):
        j = max(j, pre[i])
        dp[i] = dp[j] + 1
        k = bisect_left(prefix, prefix[i] * 2 - prefix[j])
        pre[k] = i
    return dp[n]
```

→ Complexity: O(nlog(n))

Thank's For Watching