1. Question 1

Given an array *arr* of *n* integers, the index weighted sum of a subsequence of the array formed by choosing the indices *[i1, i2, ..., ik]* is defined as *(i1 * k + a[i1]) + (i2 * k + a[i2]) ... + (ik * k + a[ik])* assuming indexing starts from 1, where *k* is the length of the subsequence. For example, for the array *arr = [1, 10, 100]*, the index weighted sum of the subsequence formed by indices [1, 3] is (1 * 2 1) + (3 * 2 + 100) = 3 + 106 = 109.

Given *arr* and an integer *max_sum*, find the length of the longest subsequence with an index weighted sum less than or equal to *max_sum*.

Example

Suppose *n* = 4, *arr* = [4, 3, 2, 1], *max_sum* = 33.

The optimal subsequence is formed by indices [1, 2, 4] i.e. values [3, 1] with the index weighted sum (1 * 3 + 4) + (2 * 3 + 3) + (4 * 3 + 1) = 7 + 9 + 13 = 29. Any subsequence of length greater than 3 will have an index weighted sum greater than *max_sum*. Hence the answer is 3.

Function Description

Complete the function *getMaxSubsequenceLen* in the editor belc

getMaxSubsequenceLen takes the following arguments:

int[n] arr: The input array

int max_sum: The maximum allowed sum of the subsequence

Returns

long int: The maximum possible length of the subsequence

2. Question 2

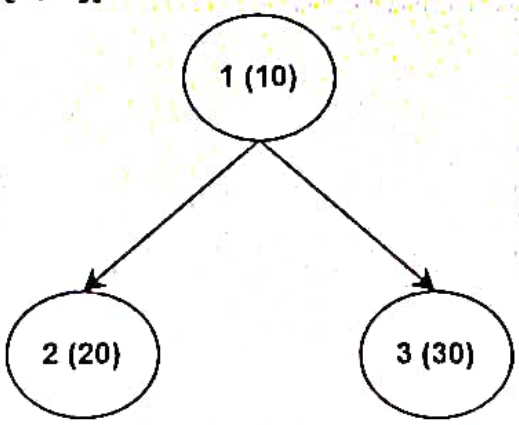
A tree is represented as an undirected connected graph of *n* nodes numbered from 1 to *n* and *n - 1* edges. The i^{th} edge connects the nodes numbered from *edges[i][0]* and *edges[i][1]* and each node *i* is associated with a value *val[i]*.

In a special move, any node can be visited from any other node directly via teleportation.

Given *val* and *edges*, find the maximum sum of values of the nodes visited on a path starting and ending at node number 1 by using the special move at most once and visiting any edge at most once.

Example

Suppose *n = 3, val = [10, 20, 30], edges = [[1, 2], [1, 3]]*



The optimal path is to traverse from node 1 to 3 and then use the special move to reach 2 and then back again to 1.

Function Description

Complete the function *getMaxValueSum* in the editor below.

getMaxValueSum takes two parameters:

int val[n]: the values associated with the nodes

int edges[n-1][2]: the edges of the tree

Returns

long int: the maximum sum of values of the nodes visited

Constraints

- $1 \leq n \leq 10^5$
- $1 \leq val[i] \leq 10^9$
- $1 \leq edges[i][0], edges[i][1] \leq n$

▶ Input Format For Custom Testing

▼ Sample Case 0

Sample Input For Custom Testing

```
STDIN           FUNCTION
-----           --------
5         →     val[] size n = 5
1         →     val = [1, 2, 3, 4,
5]
2
3
4
5
4         →     edges[] size n - 1 =
```

3. Question 3

An array *arr* is to be divided into multiple non-empty subarrays such that the total sum of costs of division is minimized. Each element of the array *arr[i]* is associated with a cost *cost[i].*

The subarrays after division are numbered 1 to *m* where *m* is the number of subarrays. For example, if *arr = [1, 2, 3, 4]* is divided into two subarrays [1, 2] and [3,4], the subarray [1, 2] is numbered 1 and [3, 4] is numbered 2.

The cost of a subarray starting from index *i* and ending at index *j* is defined as the product of value *(arr[0] + arr[1] ... + arr[j]) + (k * subarray_number)* and the sum of cost of the subarray i.e. *cost[i] + cost[i + 1] .. + cost[j].* Thus the cost of subarray from index *i* to *j* is *(arr[0] + arr[1] ... + arr[j] + k * subarray_number) * (cost[i] + ... + cost[j]).*
Here k is the subarray number factor incurred in cost.

Given *arr*, *cost* of *n* integers each and the subarray number factor k incurred in cost, find the minimum total cost of the subarrays after optimal division.

Example

Suppose *arr* = [3, 1, 4], *cost* = [2, 3, 3], and *k* = 1.

It is optimal to divide the array as [3, 1] and [4]. The cost of the first subarray will be ((3 + 1) + (1)) * (2 + 3) = 25 and that of the second subarray will be ((3 + 1 + 4) + (1 * 2)) * 3 = 30. Thus the total cost will be 25 + 30 = 55.

Function Description

Complete the function *getMinCost* in the editor below.

getMinCost takes the following arguments:

int arr[n]: the input array

int cost[n]: the cost of elements

int k: the subarray number factor incurred in cost

Returns

long int: the minimum cost of dividing the array

Constraints

- $1 \leq n \leq 2000$
- $1 \leq arr[i] \leq 1000$
- $1 \leq cost[i] \leq 1000$
- $0 \leq k \leq 1000$

Sample Input For Custom Testing

```
STDIN                FUNCTION
-----                --------
4            →       arr[] size n = 4
1            →       arr = [1 , 2 , 2 , 1]
2
2
1
4            →       cost[] size n = 4
1            →       cost = [1 , 3 , 2 , 1]
3
2
1
0            →       k = 0
```

Sample Output

```
26
```

Explanation

It is optimal to divide the array into four subarrays [1], [2], [2], and [1]. The total cost thus will be (1 + 0 * 1) * 1 + (1 + 2 + 0 * 2) * 3 + (1 + 2 + 2 + 0 * 3) * 2 + (1 + 2 + 2 + 1 + 0 * 3) * 1 = 1 + 9 + 10 + 6 = 26.