**Problem Name:** Critical Path Optimizer

**Topic:** Graph Theory, Optimization, Critical Path Analysis

**Tags:** Graph Algorithms, Topological Sorting, Dynamic Programming

**Language used:** C++

**Difficulty:** Hard

**Problem Statement:**

Given a directed acyclic graph (DAG) representing a processor network, where each node represents a processor and edges represent dependencies, calculate the minimum completion time for the network. You are allowed to replace up to k processors with high-speed versions, which reduce their processing time by half. The goal is to determine the optimal set of replacements to minimize the overall completion time.

**Input Format:**

1. An integer n representing the number of processors.
2. An integer m representing the number of dependencies.
3. An integer k representing the maximum number of processors that can be replaced with high-speed versions.
4. A list of n integers where the i-th integer represents the processing time of the i-th processor.
5. A list of m pairs of integers (u, v) where each pair represents a directed edge from processor u to processor v.

**Output Format:**

A single integer representing the minimum possible completion time of the processor network after optimally replacing up to k processors with high-speed versions.

**Constraints:**

* 1 <= n <= 10^5 (Number of processors)
* 0 <= m <= 10^5 (Number of dependencies)
* 0 <= k <= n (Maximum number of processors that can be replaced)
* 1 <= time[i] <= 10^4 (Processing time of each processor)
* The graph is a Directed Acyclic Graph (DAG).

**Sample Input 1:**

5 6 2

10 20 30 40 50

1 2

1 3

2 4

3 4

4 5

3 5

**Sample Output 1:**

85

**Explanation of Sample Input 1:**

* The graph has 5 processors and 6 dependencies.
* Initially, the critical path without any replacements is calculated.
* By replacing up to 2 processors with high-speed versions, the completion time is minimized to 85.

**Sample Input 1:**

4 4 1

15 25 35 45

1 2

1 3

2 4

3 4

**Sample Output 2:**

70

**Explanation of Sample Input 2:**

* The graph has 4 processors and 4 dependencies.
* Initially, the critical path without any replacements is calculated.
* By replacing 1 processor with a high-speed version, the completion time is minimized to 70

**Code:**

**#include <bits/stdc++.h>**

**using namespace std;**

/\*\*

\* Calculates the minimum completion time for a processor network with given high-speed replacements

\*/

int calculateCriticalPath(const vector<vector<int>>& graph, const vector<int>& time,

const vector<bool>& isHighSpeed) {

int n = graph.size() - 1; // Number of processors

// Calculate indegree for topological sort

vector<int> indegree(n + 1, 0);

for (int u = 1; u <= n; u++) {

for (int v : graph[u]) {

indegree[v]++;

}

}

// Track earliest start time for each processor

vector<int> earliestStart(n + 1, 0);

queue<int> q;

// Push all nodes with 0 indegree

for (int i = 1; i <= n; i++) {

if (indegree[i] == 0) {

q.push(i);

}

}

while (!q.empty()) {

int u = q.front();

q.pop();

// Calculate processing time based on whether the processor is high-speed

int processingTime = isHighSpeed[u] ? time[u] / 2 : time[u];

for (int v : graph[u]) {

// Update earliest start time for the dependent processor

earliestStart[v] = max(earliestStart[v], earliestStart[u] + processingTime);

// Reduce indegree and add to queue if all dependencies are processed

indegree[v]--;

if (indegree[v] == 0) {

q.push(v);

}

}

}

// Calculate project completion time (maximum finish time)

int result = 0;

for (int i = 1; i <= n; i++) {

int processingTime = isHighSpeed[i] ? time[i] / 2 : time[i];

result = max(result, earliestStart[i] + processingTime);

}

return result;

}

/\*\*

\* Finds the optimal critical path by iterative refinement

\*/

int optimizeCriticalPath(const vector<vector<int>>& graph, const vector<int>& time, int k, int n) {

vector<bool> isHighSpeed(n + 1, false);

// Initial critical path calculation without any high-speed processors

int bestTime = calculateCriticalPath(graph, time, isHighSpeed);

// Iteratively improve by replacing the most impactful processor

for (int iteration = 0; iteration < k; iteration++) {

int bestProcessor = -1;

int bestTimeReduction = 0;

// Try replacing each processor that's not already high-speed

for (int i = 1; i <= n; i++) {

if (!isHighSpeed[i]) {

isHighSpeed[i] = true;

int newTime = calculateCriticalPath(graph, time, isHighSpeed);

int timeReduction = bestTime - newTime;

if (timeReduction > bestTimeReduction) {

bestTimeReduction = timeReduction;

bestProcessor = i;

}

isHighSpeed[i] = false; // Reset for next iteration

}

}

// If no improvement is possible, break

if (bestTimeReduction <= 0) {

break;

}

// Replace the processor that gives the best improvement

isHighSpeed[bestProcessor] = true;

bestTime -= bestTimeReduction;

}

return bestTime;

}

int main(int argc, char\* argv[]) {

// Allow input from a file if provided

if (argc > 1) {

freopen(argv[1], "r", stdin); // Redirect input from the given file

}

// Fast I/O

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

int n, m, k;

cin >> n >> m >> k;

vector<int> time(n + 1);

for (int i = 1; i <= n; i++) {

cin >> time[i];

}

vector<vector<int>> graph(n + 1);

for (int i = 0; i < m; i++) {

int u, v;

cin >> u >> v;

graph[u].push\_back(v);

}

int result = optimizeCriticalPath(graph, time, k, n);

cout << result << endl;

return 0;

}