SOI 2P 2020

Solution Booklet

Swiss Olympiad in Informatics

13–16 March 2019

Candy distribution

Task Idea	Monika Steinova
Task Preparation	Monika Steinova
Description English	Monika Steinova
Description German	Jan Schär
Description French	Florian Gatignon
Solution	Monika Steinova

Subtask 1: 50 Points (simulation)

The task can be solved by a straightforward greedy simulation: Unless the three candy counts are not equal, add a candy to the two smallest counts.

To see why this process terminates and that it yields the minimum number of rounds we observe how candy counts change during the simulation.

Let (a, b, c) be the candy counts of mice A, B, C. WLOG let $a \leq b \leq c$. In the first $c - b$ rounds our simulation adds $c - b$ candy to mouse A and $c - b$ candy to mouse B yielding candy counts $(a + c - b, b + c - b, c) = (a + c - b, c, c)$. (Notice that since $b \geq a$, the first element $a - b + c \leq c$.)

Let's observe how the candy counts change from now on after every two rounds. As long as mouse A has less candy than her sisters we give 2 candy to A and 1 candy to each of B and C. Hence after $2k$ such rounds the candy counts are $(a + c - b + 2k, c + k, c + k)$.

As the A's candy count $c + (a - b) \leq c$ after $k := b - a \geq 0$ rounds the candy counts will be equal: $(a+c-b+(b-a)+(b-a), c+(b-a), c+(b-a)) = (c+k, c+k, c+k)$ and our simulation terminates.

During the whole simulation the candy count of A was less than candy count of C. Hence the value $k := b - a$ is the minimal one needed to reach the equality of candy counts.

Overall we did $(c-b)+2k = b+c-2a$ rounds and hence the time complexity is $O(\max(a, b, c))$.

```cpp
#include <iostream>
#include <vector>
#include<algorithm>

using namespace std;

int main() {
    long long round = 0;
    vector<long long> C(3, 0);
    cin >> C[0] >> C[1] >> C[2];

    while( !( (C[0] == C[1]) && (C[1] == C[2]) ) ) {
        sort(C.begin(), C.end());
        ++C[0];
        ++C[1];
        ++round;
    }
    cout << round << endl;
    return 0;
}
```

Subtask 2: 100 Points in a constant time

In the previous section we have proven that $b + c - 2a$ rounds are necessary and sufficient. Hence our algorithm can just output this number and terminate.

There exists another approach which doesn't need to analyze rounds. Observe that giving one candy to two different mice is equivalent to taking one candy from the third mouse. You can prove this by observing how the differences between mices change if the two candies are added. Hence the task can be reformulated to "What is the minimum amount of candy that has to be removed from a mouse to make the candy counts of the three mice equal?" And the answer to this is simple. It is $(b-a)+(c-a)=b+c-2a$ $(a \leq b \leq c)$.

```cpp
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

int main(void) {
    vector<long long> C(3, 0);
    cin >> C[0] >> C[1] >> C[2];
    sort(C.begin(), C.end());
    cout << C[1] + C[2] - 2*C[0] << endl;
    return 0;
}
```

Workshops

Task Idea	Johannes Kapfhammer
Task Preparation	Florian Gatignon
Description English	Florian Gatignon
Description German	Martin Raszyk
Description French	Florian Gatignon
Solution	Florian Gatignon

Subtask 1: 20 points (Bruteforce)

In the first subtask, the limits were low, so a simple bruteforce was enough to get those first 20 points. The idea is to compute whether you can organize 1 workshop, then 2, and so on until you find a number of workshops that you can't reach.

To determine whether it is possible to organize x workshops, we need to check how many jokers are needed for each job and compare the sum of those values to J. Another way to do it is to always check whether it is possible to organize one workshop, but removing the needed number of jokers from J afterwards. This is how the solution below works. For each organizable workshop, we use $O(N)$ time, so $O(NW)$ in total if the answer is W. The answer itself can't be larger than $J + \min_{0<=i<N} b_i$.

```cpp
#include <bits/stdc++.h>
#define ll long long
using namespace std;
ll N, J;
vector<ll> a, b;

int main() {
	ios_base::sync_with_stdio(false);
	cin.tie(0);
	cin >> N >> J;
	a.resize(N); b.resize(N);
	for(ll i = 0; i < N; i++) cin >> a[i];
	for(ll i = 0; i < N; i++) cin >> b[i];
	ll ans, tot=0;
	for(ans=-1;tot<=J; ans++) {
		J-=tot;
		tot = 0;
		for(ll i = 0; i < N; i++) {
			if(a[i]>b[i]) {
				tot += a[i]-b[i];
				b[i] = 0;
			} else {
				b[i] -= a[i];
			}
		}
	}
	cout << ans << endl;
}
```

Subtask 2: 30 points (No jokers)

In the second subtask, there were more jobs and regular volunteers but there weren't any jokers available. You just had to compute how many workshops could be organized with normal volunteers. To do so was quite easy: the answer is $\min_{0<=i<n}(\lfloor \frac{a_i}{b_i} \rfloor)$, which one may compute in $O(N)$.

Of course, you can combine this solution with that of Subtask 1 to get a total of 50 points.

```
#include <bits/stdc++.h>
#define ll long long
using namespace std;
ll N, J;
vector<ll> a, b;

int main() {
	ios_base::sync_with_stdio(false);
	cin.tie(0);
	cin >> N >> J;
	a.resize(N); b.resize(N);
	for(ll i = 0; i < N; i++) cin >> a[i];
	for(ll i = 0; i < N; i++) cin >> b[i];
	ll ans=1000000000000;
	for(ll i = 0; i < N; i++) {
		ans = min(ans,b[i]/a[i]);
	}
	cout << ans << endl;
}
```

Subtask 3: 100 points (Binary search)

In the third and last subtask, all limits were quite large. For all $0 <= x <= W$, it is possible to organize x workshops, and for all $x > W$ it is impossible. In other words, we are looking for a *tipping point*. That means that we don't need to test every possible number of workshops until W, we can binary search for that tipping point.

Basically, we take a large interval that includes all possible answers given the limits. We take the middle of the interval; if we can organize that many workshops, we keep that value in memory as the best answer until now and recurse on the upper half of the interval; else, we look at the lower half. Each time, we can check in $O(N)$ whether it is possible to organize a certain amount of workshops (in a trivial way), so our total running time is $O(N \log(J + \min_{0<=i<N} b_i))$.

```
#include <bits/stdc++.h>
#define ll unsigned long long
using namespace std;
ll N, J;
vector<ll> a, b;

bool check(ll x) {
	ll tot=0ll;
	for(ll i = 0; i < N; i++) {
		if(x*a[i]>b[i])
			tot += x*a[i]-b[i];
	}
	return tot<=J;
}

int main() {
	ios_base::sync_with_stdio(false);
	cin.tie(0);
	cin >> N >> J;
	a.resize(N); b.resize(N);
	for(ll i = 0; i < N; i++) cin >> a[i];
	for(ll i = 0; i < N; i++) cin >> b[i];
	ll lo = 0ll, hi = 2134567890, ans = 0ll;
	while(lo<=hi) {
		ll mi = (lo+hi)/2;
		if(check(mi)) {lo = mi+1; ans = mi;}
		else hi = mi-1;
	}
	cout << ans << endl;
}
```

Toll Roads

Task Idea	Daniel Rutschmann
Task Preparation	Martin Raszyk
Description English	Martin Raszyk
Description German	Martin Raszyk
Description French	Erwan Serandour
Solution	Johannes Kapfhammer

Given a graph with edge weights 0 or 1, the task was to answer multiple queries that asked you to compute the distance between a pair of vertices.

Starting from the basic solution (one Dijkstra or one 0-1-BFS), there are two independent observations required for 100 points: precomputing distances to answer the queries faster and compressing 0-edges.

One Dijkstra per Query (25 points)

The distance between two vertices can be computed using Dijkstra. The graph is the same as the input graph, where the private highway segments have cost 1 and the state highway segments have cost 0. Note that Djikstra also works for graphs where the edge cost is 0 (Dijkstra only breaks if the costs can be negative). Since the weight corresponds to the cost, the shortest path will be the cheapest one.

```cpp
#include <bits/stdc++.h>
using namespace std;

const int INF = 1e9;

// returns the distances from start to all other vertices
vector<int> dijkstra(vector<vector<pair<int, int>>> const& g, int start) {
  vector<int> dist(g.size(), INF);
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
  pq.emplace(0, start);
  while (!pq.empty()) {
    auto [cost, v] = pq.top();
    pq.pop();
    if (dist[v] != INF) continue;
    dist[v] = cost;
    for (auto [u, c] : g[v])
      pq.emplace(cost+c, u);
  }
  return dist;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(0);

  int n, m, k, q;
  cin >> n >> m >> k >> q;

  vector<vector<pair<int, int>>> g(n);
  for (int i=0; i<m; ++i) {
    int a, b; cin >> a >> b;
    g[a].emplace_back(b, 0);
    g[b].emplace_back(a, 0);
  }
  for (int i=0; i<k; ++i) {
    int a, b; cin >> a >> b;
    g[a].emplace_back(b, 1);
    g[b].emplace_back(a, 1);
  }
  for (int i=0; i<q; ++i) {
    int a, b; cin >> a >> b;
    cout << dijkstra(g, a)[b] << '\n';
  }
}
```

This solution runs in $O(q \cdot (m+k)\log(m+k))$.

Once Dijkstra per Vertex (50 points)

It turns out we don't need to compute one Dijkstra per query. A run of Dijkstra starting at vertex s computes the distances $d(s,v)$ for *all* other vertices v. Therefore, we can precompute the distances for all start vertices s to all other vertices.

The only change required is replacing the query loop with the following:

```
vector<vector<int>> dist(n);
for (int s=0; s<n; ++s)
  dist[s] = dijkstra(g, s);
for (int i=0; i<q; ++i) {
  int a, b; cin >> a >> b;
  cout << dist[a][b] << '\n';
}
```

This reduces the running time from $O(q \cdot (m+k)\log(n))$ to $O(n \cdot (m+k)\log(m+k)+q)$.

Compressing components of 0-edges (50 points)

Until now, we didn't make use of the fact that all distances are 0 or 1. Looking at the limits, most edges have weight 0.

If two vertices a and b are connected with an edge of weight 0, we can just pretend they are the same vertex. Why? Because we have $d(a,v)=d(b,v)$ for all vertices v. (Proof: in any graph we have the triangle inequality $d(a,b) \le d(a,x)+d(x,b)$. Applying this to a, b and v and noting $d(a,b)=0$: $d(a,v) \le d(a,b)+d(b,v)=d(b,v)$ and $d(b,v) \le d(b,a)+d(a,v)=d(a,v)$, so $d(a,v)=d(b,v)$.)

More generally, this applies to all pairs of vertices that have distance 0. So we look for "clumps" of vertices that have distance 0 and compress them to one mega-vertex.

This is fast because we have to compress the vertices just once, and then then all remaining edges are the ones with distance 1.

To compress, we create a graph containing only the edges of cost 0. We identify components using DFS and save for each vertex to which component it belongs. Then, we build a second graph where the vertices are components of the first graph, containing only edges of cost 1.

Precomputing can be done in $O(n+m)$. Then, to answer queries one could run one BFS per query and get $O(n+m+q \cdot k)$

```
#include <bits/stdc++.h>
using namespace std;

// sets component[w]=comp_id for all vertices w that are reachable from v
// assuming none of those vertices have been visited yet
void dfs(vector<vector<int>> const& g, int v, int comp_id, vector<int>& component) {
  if (component[v] != -1) return;
  component[v] = comp_id;
  for (auto w : g[v])
    dfs(g, w, comp_id, component);
}

// returns the number of components and the component id of each vertex
pair<int, vector<int>> compress(vector<vector<int>> const& g) {
  vector<int> component(g.size(), -1);
  int comp_id = 0;
  for (int i=0; i<(int)g.size(); ++i)
    if (component[i] == -1)
      dfs(g, i, comp_id++, component);
  return {comp_id, component};
}

// returns the distances from start to all other vertices
vector<int> bfs(vector<vector<int>> const& g, int start) {
  vector<int> dist(g.size(), -1);
  queue<int> q;
  dist[start] = 0;
  q.push(start);
  while (!q.empty()) {
    auto v = q.front();
    q.pop();
    for (auto w : g[v]) {
      if (dist[w] == -1) {
        dist[w] = dist[v] + 1;
        q.push(w);
      }
    }
  }
  return dist;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(0);

  int n, m, k, q;
  cin >> n >> m >> k >> q;

  vector<vector<int>> g0(n); // graph of 0-edges
  for (int i=0; i<m; ++i) {
    int a, b; cin >> a >> b;
    g0[a].push_back(b);
    g0[b].push_back(a);
  }
  auto [num_components, component] = compress(g0);
  vector<vector<int>> g1(num_components); // graph of 1-edges
  for (int i=0; i<k; ++i) {
    int a, b; cin >> a >> b;
    a = component[a];
    b = component[b];
    if (a != b) {
      g1[a].push_back(b);
      g1[b].push_back(a);
    }
  }
  for (int i=0; i<q; ++i) {
    int a, b; cin >> a >> b;
    a = component[a];
    b = component[b];
    cout << bfs(g1, a)[b] << '\n';
  }
}
```

Compressing and Precomputing (100 points)

Both ideas can be combined for 100 points.

Again, the only part that changes is that we precompute the table of all distances before answering the queries:

```
vector<vector<int>> dist(num_components);
for (int s=0; s<num_components; ++s)
  dist[s] = bfs(g1, s);
for (int i=0; i<q; ++i) {
  int a, b; cin >> a >> b;
  cout << dist[component[a]][component[b]] << '\n';
}
```

With this, the running time is $O(n + m + n \cdot k + q)$.

Alternative idea: 0-1 BFS

Instead of using Dijkstra, one could do a so-called 0-1-BFS. This is a BFS that works if the graph has edges of weights 0 and 1 and runs in $O(n + m)$ (where m is the number of total edges) instead of $O(n + m \log n)$ of a normal Dijkstra (and $O(n \log n + m)$ for a Dijkstra with a Fibonacci Heap).

The algorithm is explained here: `https://cp-algorithms.com/graph/01_bfs.html`.

Alternative idea: Union Find

In the code above the components are identified using DFS. Of course, one could also use union find to handle this. Union find is more general because it can handle dynamic updates. This was not needed in this task, however.

Pizzacut

Task Idea	Daniel Rutschmann
Task Preparation	Daniel Rutschmann
Description English	Daniel Rutschmann
Description German	Benjamin Schmid
Description French	Florian Gatignon
Solution	Daniel Rutschmann

Greedy doesn't work

There are various wrong greedy solutions, for instance based on giving each mouse the minimum number of grams of pizza. In fact, most of the already fail on the samples. While one can show that some these solutions use at most one pizza too many (see 90p/100p solution), there is no easy way of fixing this. In this task, the limits for A and B are quite small. This is a strong indicator against greedy solutions, as those would be fast enough even for large A and B.

30 Points

Let's use "A-slices" for slices of A grams and "B-slices" for slices of B grams. The small limits for A and B suggest some solution with dynamic programming. In this subtask, we can use $DP[i][x]$ = "minimum number of B-slices we need to use to feed the first i mice if we use exactly x A-slices". When considering the i-th mouse, we try all possible ways of efficiently feeding this mouse. More precisely, we iterate over the number of A-slices this mouse gets and compute the minimum number of B-slices this mouse then needs.

To bound the runtime, note that each mouse needs at most $2B$ A-slices, as $g_i \leq 2AB$, so x is at most $2NB$. Hence the above DP has $O(N^2B)$ states. Each state takes $O(B)$ times to compute, as there are $O(B)$ ways of efficiently feeding a fixed mouse, so we get a runtime of $O(N^2B^2)$.

```cpp
#include <bits/stdc++.h>
using namespace std;

constexpr int inf = 1e9;

// ceil(a/b)
int cdiv(int a, int b){
    if(a > 0) return (a + b-1)/b;
    return a/b;
}
template<typename T>
void xmin(T&a, T const&b){
    a = min(a, b);
}
signed main(){
    int n, A, B;
    cin >> n >> A >> B;
    vector<int> v(n);
    for(auto &e:v) cin >> e;
    const int X = 2*n*N+1;
    vector<vector<int> > dp(n+1, vector<int> (X, inf));
    dp[0][0] = 0;
    for(int i=0; i<n; ++i){
        const int e = v[i];
        for(int a=0; a <= e/A+1; ++a){
            const int b = cdiv(max(0, e-a*A), B);
            for(int x = 0; x < X-a; ++x){
                xmin(dp[i+1][x+a], dp[i][x] + b);
            }
        }
    }
    int ans = inf;
    for(int i=0; i<X; ++i){
        xmin(ans, cdiv(dp[n][i], A) + cdiv(i, B));
    }
    cout << ans << "\n";
}
```

45 Points

To get 45 Points, we need to handle larger values of g_i. We make the following observation: If a mouse eats more than $2AB$ grams of pizza, then it eats at least AB grams worth of A-slices or at least AB grams worth of B-slices, i.e. a full pizza of A-slices or a full pizza of B-slices. It makes no difference whether this full pizza is cut into A-slices or B-slices, so we can just give this mouse a full pizza and reduce g_i by AB. This allows us to assume that $g_i \leq 2AB$, at which point we can use the previous DP solution.

70 points

As in the solution for 45 Point, we reduce to the case where $g_i \leq 2AB$. In the 30 points solution, we used $DP[i][x]$ where x was the total number of A-slices we used, so x could get as large as $2BN$. Since we only care about the number of pizzas and not about the exact number of A-slices or B-slices, we could replace a whole pizza of A-slices by a whole pizza of B-slices. In other words, if $x \geq B$, then we can reduce x by B if we increase the number of B-slices of A. This allows us to only consider $0 \leq x < B$, which reduces the number of DP states to $O(NB)$. This solution runs in $O(NB^2)$.

```cpp
#include <bits/stdc++.h>
using namespace std;

constexpr int inf = 1e9;

int cdiv(int64_t a, int64_t b){
    if(a > 0) return (a + b-1)/b;
    return a/b;
}

template<typename T>
void xmin(T&a, T const&b){
    a = min(a, b);
}

signed main(){
    int n, A, B;
    cin >> n >> A >> B;
    vector<int> v(n);
    int64_t whole = 0;
    for(auto &e:v) {
        cin >> e;
        if(e > 2*A*B){
            int x = e/(A*B)-2;
            e -= x*A*B;
            whole += x;
        }
    }
    vector<vector<int64_t> > dp(n+1, vector<int64_t>(B, inf));
    dp[0][0] = 0;
    for(int i=0; i<n; ++i){
        const int e = v[i];
        for(int a=0; a <= min(B, e/A+1); ++a){
            const int b = (max(0, e-a*A)+B-1)/B;
            for(int x = 0; x < B-a; ++x){
                xmin(dp[i+1][x+a], dp[i][x]+b);
            }
            // whole pizza of A-slices -> whole pizza of B-slices
            for(int x = B-a; x < B; ++x){
                xmin(dp[i+1][x+a-B], dp[i][x]+b + A);
            }
        }
    }
    int ans = inf;
    for(int i=0; i<B; ++i){
        xmin(ans, cdiv(dp[n][i], A) + cdiv(i, B));
    }
    cout << whole+ans << "\n";
}
```

90 / 100 points

Getting more that 70 points is very difficult. The basic idea is to first start with some greedy solution and then do a DP to see if this greedy solution can be improved. Starting with a greedy solution greatly reduces the number of values the DP can attain. This allows us to use some data structure to quickly decide which options improve our DP and which don't.

Concretely, we first precompute all numbers of grams we can reach with A-slices and B-slices. Then for each mouse, we give that mouse the fewest number of grams of pizza possible. This gives us some number of A-slices and some number of B-slices we use, so we order enough pizzas to get that many slices.

We claim that this uses at most one pizza too many. To show this claim, note that this greedy solution uses the minimum number of grams of pizza. The only inefficiency comes from the fact that number of A-slices we use might be not divisible by B, in which case we have some A-slices that we don't use. We have at most $(B-1)$ leftover A-slices and at most $(A-1)$ leftover B-slices, so the number of grams of pizza we order but don't use is less than $2AB$. Hence any other solution can save strictly less than $2AB$ grams of pizza, so it can save at most one whole pizza. Based on this observation, there are a few different ways of getting an $O(n + AB^2)$ solution. We will first discuss the original solution and then an easier solution that we only discovered after the contest.

Original solution

Starting from the greedy solution, we can try to get a better solution by give some mice more A-slices and fewer B-slices or vice-versa. This results in the following: $DP[i][\Delta a]$ = "minimum number of additional B-slices we need if we use Δa additional A-slices for the first i mice" (this will be zero or negative). Initially, $DP[0][\Delta a] = 0$. When considering the i-th mouse, we again try all efficient ways of feeding it and check how this would change the greedy solution. As in the 70p solution, we again use the trick of replacing whole pizzas of A-slices by whole pizzas of B-slices to assume $0 \leq \Delta a < B$.

This solution still runs in $O(nB^2)$, so it will only give 70 points. To speed it up, note that $DP[i+1][\Delta a] \leq DP[i][\Delta a]$, so this is never positive and that $DP[i][\Delta a] > -3A$ as we can't improve the greedy solution by more that one pizza. Hence $DP[i][\Delta a]$ only decreases as i increases and it can decrease at most $O(A)$ times.

When considering the i-th mouse, each change will be of the form: If we use x additional A-slices, then we need y additional B-slices, where either $x > 0, y < 0$ or $x < 0, y > 0$. The key idea is to use some data structure to maintain for each x $(-B \leq x \leq B)$ the minimum y that would cause a decrease in the DP. This data structure should maintain a multiset of B values (one value for each possible value of Δa), each between $-2A$ and $+2A$ and it should support finding the maximum and updating a value in the set in amortized $O(1)$. This can be achieved by storing a vector of counts and the current maximum. Whenever we compute a new entry of our DP table, we need to update this datastructure.

This data structure allows us to quickly (in $O(1)$) decide whether a change (x, y) will improve the DP or not. As the DP can be improved at most $O(AB)$ times, we spend $O(AB^2)$ time updating the DP in total. We need $O(nB)$ time to consider all possible changes, so we get a total runtime of $O(nB + AB^2)$. This scores 90 points.

To get 100 points, we note that there are at most $2AB$ different values of g_i. If some value of g_i doesn't change the DP, we can discard all mice with that value. Hence for every Mouse we consider, we either throw away some g_i, or we find some improvement in the DP. Hence we consider only $O(AB)$ mice, so the runtime is $O(n + AB^2)$.

Easier solution (Thanks Fabian!)

We again start with the greedy solution and try to improve it by some changes of the form: Give some mouse x additional A-slices and y additional B-slices where one of x and y is positive and the other is negative. We make two observations.

1. Whether or not some change (x, y) with $x < 0$ is possible for some mouse only depends on the the number of A-slices this mouse has in the greedy solution. More precisely, this mouse needs to have at least x A-slices. The same also holds for changes with $y < 0$ and B-slices.
2. There is an optimal solution that does at most B such changes.

To see (2), we use the following easy lemma: Given B integers $x_1, \ldots, x_B$, there is a non-empty subset with sum divisible by B. The proof of this is an easy exercise (Hint: look at prefix sums modulo B). Suppose now that we do B changes $(x_1, y_1), (x_2, y_2), \ldots, (x_B, y_B)$. By the lemma, there is a subset of changes whose x_i is divisible by B. Let's remove these changes. This does not change the number of A-slices modulo B and it reduces the number of changes. As a change can only increase the total number of grams of pizza we use, removing these changes only improves our solution. Hence we can always get an optimal solution with fewer than B changes.

With these observations, we see that we only need to do changes on the B mice with a largest number of A-slices and the B mice with a largest number of B-slices. All other mice will get exactly the slices they would get in the greedy solution. Hence we can run the 70p DP on these $2B$ mice and the greedy on all other mice. This reduces n to $2B$, so the runtime is $O(n + B^3)$.

```
#include <bits/stdc++.h>
using namespace std;

constexpr int inf = 1e9;
template<typename T>
void xmin(T&a, T const&b){
    a = min(a, b);
}
int cdiv(int a, int b){
    if(a > 0) return (a + b-1)/b;
    return a/b;
}
struct Greedy_Decomp{
    Greedy_Decomp(int A_, int B_) : A(A_), B(B_), U(2*A*B+1), pre(U, {-1, -1}){
        for(int i=0; i <= min(B-1, U/A); ++i){
            for(int j=0; j <= U/B; ++j){
                const int r = i*A + j*B;
                if(r < U){
                    pre[r] = make_pair(i, j);
                }
            }
        }
        for(int i=U-1; i>=0; --i){
            if(pre[i].first == -1) pre[i] = pre[i+1];
        }
    }
    pair<int, int> decompose(int x) const {
        return pre[x];
    }
    int A, B, U;
    vector<pair<int, int> > pre;
};
struct Mouse{
    int a, b, hunger;
};
signed main(){
    cin.tie(0);
    ios_base::sync_with_stdio(false);
    int n, A, B;
    cin >> n >> A >> B;
    Greedy_Decomp greed(A, B);
    vector<int> v(n);
    int64_t whole = 0;
    for(auto &e:v) {
        cin >> e;
        if(e > 2*A*B){
            int x = e/(A*B)-1;
            e -= x*A*B;
            whole += x;
        }
    }
    vector<Mouse> mice(n);
    for(int i=0; i<n; ++i){
        auto [a,b] = greed.decompose(v[i]);
        mice[i].hunger = v[i];
        mice[i].a = a;
        mice[i].b = b;
    }
    vector<Mouse> used_mice;
    if(n <= 2*B){
        used_mice.swap(mice); // use all mice
    } else {
        // use B mice with largest number of A-slices
        nth_element(mice.begin(), mice.begin()+B, mice.end(),
            [](Mouse const&x, Mouse const&y){return x.a > y.a;});
        used_mice.insert(used_mice.end(), mice.begin(), mice.begin()+B);
        mice.erase(mice.begin(), mice.begin()+B);

        // use B mice with largest number of B-slices
        nth_element(mice.begin(), mice.begin()+B, mice.end(),
            [](Mouse const&x, Mouse const&y){return x.b > y.b;});
        used_mice.insert(used_mice.end(), mice.begin(), mice.begin()+B);
        mice.erase(mice.begin(), mice.begin()+B);
    }
    // now run 70p dp on the 2B best mice
    const int k = used_mice.size();
    vector<vector<int> > dp(k+1, vector<int>(B, inf));
    dp[0][0] = 0;
    for(int i=0; i<k; ++i){
        const int e = used_mice[i].hunger;
        for(int a=0; a <= min(B, e/A+1); ++a){
            const int b = (max(0, e-a*A)+B-1)/B;
            for(int x = 0; x < B-a; ++x)
                xmin(dp[i+1][x+a], dp[i][x]+b);
            // whole pizza of A-slices -> whole pizza of B-slices
            for(int x = B-a; x < B; ++x)
                xmin(dp[i+1][x+a-B], dp[i][x]+b + A);
        }
    }
    // run greedy on other mice
    int total_a = 0, total_b = 0;
    for(auto const&e:mice){
        total_a+=e.a;
        total_b+=e.b;
    }
    int ans = inf;
    for(int i=0; i<B; ++i){
        xmin(ans, cdiv(total_b + dp[k][i], A) + cdiv(total_a + i, B));
    }
    cout << whole+ans << "\n";
}
```