Solution — From Russia with Love

1 Modelling

We are given an array of numbers $x_0, \ldots, x_{n-1}$, which represent the values of the n coins placed on a line in this order. We consider the following game: m players $p_0, \ldots, p_{m-1}$ take turns to pick a coin from one end of the line and to take that coin. This coin is then added to the personal pile of the player that took it. The game continues until there are no more coins left. The *winnings* of a player is the sum of the values of all the coins in his/her pile at the end of the game. We are asked to find the maximum winnings that player p_k is guaranteed to get if he/she plays optimally, regardless of how the other players play.

More formally, at any point during the game, the available coins are a subset of the original set of coins of the form $x_i, \ldots, x_{j-1}$ for some $0 \leqslant i \leqslant j \leqslant n$. Notice in particular that all the indices of the available coins are always consecutive, that is, they form an interval. This can be seen by induction. At the beginning of the game, the indices form the interval $[0, n]$. At each turn, either the coin with the smallest index, or the coin with the largest index is removed from the line. Therefore, the indices of the available coins remain consecutive, as claimed.

We use some game theoretic notions to think about this problem. A *game graph* is a directed acyclic graph G in which each vertex u represents a possible state of the game $S(u)$, and there is an edge from vertex u to vertex v if the game can transition from state $S(u)$ to state $S(v)$ in one turn of whichever player's turn it is to play. The state of our game is determined by which coins have not been taken yet and whose turn it is to play. As discussed above, the set of coins that have not been taken yet can be characterised by two integers, i and j, with $0 \leqslant i \leqslant j \leqslant n$.

There are m options for whose turn it is to play. However, notice that each player takes exactly one coin at each turn. So, if we know how many coins are left in the game, we can uniquely determine whose turn it is. Player p_0 plays first, when there are n coins available. Next, there are $n-1$ coins available and it is player p_1's turn, etc. If there are $n-m$ coins available, it is again p_0's turn. In general, if there are c coins available, it is player $p_{(n-c) \bmod m}$'s turn. Thus, if our available coins are $x_i, \ldots, x_{j-1}$, then $c = j - i$, so it is player $p_{(n-j+i) \bmod m}$'s turn. Since which player's turn it is can be uniquely determined by which coins are available, all we need to describe the state of the game are our two integers i and j. As they can both take values between 0 and n, there are $\Theta(n^2)$ possible states of the game, and thus $\Theta(n^2)$ vertices in our game graph G. From now on we represent a state of the game as an ordered pair (i, j).

The transitions between states of the game are represented by the edges of G. There is an edge in G from vertex u with $S(u) = (i, j)$ to vertex v with $S(v) = (i', j')$ if one of these holds:

1. The next player takes coin x_{j-1}, so $i' = i$ and $j' = j - 1$.
2. The next player takes coin x_i, so $i' = i + 1$ and $j' = j$.

The game is over when there are no available coins, that is, when we are in a state (i, j) with $i = j$. Thus, each vertex in the game graph has two outgoing edges, except for the leaves, which

have no outgoing edges, and except for the cases (i, j) with $j = i + 1$, which have one outgoing edge. The number of edges in our game graph $E(G)$ is thus also $\Theta(n^2)$.

2 Algorithm Design

We are now interested in the fate of player p_k. Notice that we are looking for the winnings that p_k is *guaranteed* to have if he/she plays optimally, regardless of how the others play. This means that we cannot assume anything about the behaviour of the other players—for instance, they will not necessarily attempt to maximize their own winnings. Thus, we have to consider the worst case scenario from the point of view of player p_k, and see what he/she can get than if he/she plays optimally.

Why greedy does not work. *This is a good opportunity to take a moment and see why a number of greedy approaches that may seem intuitive do not work.*

In general, the intuitive reason why in most cases greedy approaches do not work for problems with games is that greedy strategies choose the move that yields the best result for the moment, but this is usually not the same as making a move that results in the best outcome in the long run. In this case, taking a coin that has a very high value may seem like the best option, but it might lead to the game taking an unfavourable direction. Each move a player makes moves the game along an edge in the game graph to a new vertex. From this new vertex, only a subgraph of the original game graph is reachable. Thus, choosing what move to make is equivalent to choosing the best combination of value of the coin we take now and value that we are guaranteed to get out of the game subgraph that we lock ourselves into.

Let us consider an example for this intuition. Suppose there are two players and suppose the $n = 4$ available coins have values $x_0 = 3, x_1 = 2, x_2 = 5, x_3 = 4$. Suppose it is our turn to play. If we play greedily and always pick a coin with the highest value, we would pick x_3. Notice that this means that the other player may pick x_2, leaving us to choose between x_0 and x_1, in which case our winnings are at most $x_0 + x_3 = 3 + 4 = 7$. It would have been better if we had picked x_0 instead of x_3 at the beginning, because then no matter what the other player does, we would be able to pick x_2, yielding a total winnings of $x_0 + x_2 = 3 + 5 = 8$. Thus, to find the best move we should think ahead into the future. This explains why the strategy to always choose a coin of highest value is *not* optimal in general.

The reason why we cannot assume anything about the behaviour of the other players is very similar. We cannot assume that they are "nice" to player k and always take the coin with smallest value because we need to determine how much the k-th player can win *regardless* of what the other players do. Thus, we have to consider the worst-case scenario for player k. The next natural idea might be to assume that the other players always pick the coin with maximum value, thinking that this would lead to a worst-case scenario for player k.

But this is not true: A greedy strategy of the other players does *not* necessarily lead to a worst-case scenario for player k. To see this, we can again look at the same example as above, but turn it upside down: suppose there are two players and $x_0 = 3, x_1 = 2, x_2 = 5, x_3 = 4$. Suppose it is the other player's turn now. If he/she plays greedily and takes $x_3 = 4$, we can get $x_2 = 5$, and then the other players would (again greedily) take $x_0 = 3$, leaving us with winnings of $x_2 + x_1 = 5 + 2 = 7$. On the other hand, if the other player starts by taking $x_0 = 3$ as a first turn, we can only get $x_1 + x_3 = 2 + 4 = 6$ as no matter what we do, the other player will then

take $x_2 = 5$. Thus, if the other player thinks about the future a little bit, he/she can put us in a worse situation than if he/she plays greedily. But what does it mean to think about the future?

The recursive relation *The way to "think about the future" is dynamic programming, which implicitly considers all possible turns of events (or paths through the game graph), without actually considering them.*

In order to pick an optimal strategy, we need to consider all possible scenarios of how the game may develop, and pick the sequences of actions that guarantee to lead to the most favourable outcome. But there are a whole lot of possible scenarios how the game may develop. Since each player usually has two options at each turn (except when taking the last coin)—either to take the leftmost coin or the rightmost coin—and the number of coins to be taken is n, we get that there are 2^{n-1} ways in which the game may develop. A time complexity of $\Theta(2^n)$ is already infeasible for $n \approx 40$, let alone $n \approx 10^3$. Thus we cannot afford to explicitly consider all possible scenarios. This is where dynamic programming comes in and allows us to implicitly consider all of these scenarios efficiently.

The reason we can apply dynamic programming is because this problem exhibits the *optimal subproblems* structure. That is, a state of the game (a vertex in the game tree) can lead to several different states, depending on what the player does, and each of these new states themselves are subproblems that need to be solved optimally in order for the problem as a whole to be solved optimally. The key idea is to notice that whenever we are in a state of the game, in which it is the k-th player's turn, we have to choose the move that leads to the maximum winnings for the k-th player; and whenever we are in a state in which it is some other player's turn, we have to choose the move that leads to the minimum winnings for the k-th player. The second part of this statement is true because we are looking for the winnings that the k-th player is *guaranteed* to get—so we are looking for the smallest possible winnings for player k given he/she plays optimally.

We can associate with each vertex v of our game graph a number $w(v)$, which is the maximum winnings player k is guaranteed to get from this moment on in the game (that is, given that the game starts in state $S(v)$). Then if it is the k-th player's turn, we have that

$$w(v) = \max_{u,(v,u)\in E(G)} \big(x_h + w(u)\big),$$

where x_h is the coin that is removed to get from $S(v) = (i, j)$ to $S(u) = (i', j')$, namely either $h = i$ or $h = j - 1$. That is, for any possible turn that the k-th player can do (i.e. taking the h-th coin), we evaluate the winnings he/she would get in that scenario (the sum of the value of the h-th coin and the winnings from that point onwards, $w(u)$).

On the other hand, if it is some other player's turn, we have

$$w(v) = \min_{u,(v,u)\in E(G)} w(u),$$

because, as discussed above, we have to consider the worst-case scenario for player k. In the end, we are interested in $w(a)$, where a is the starting vertex of the game graph corresponding to the full interval of coins, that is $S(a) = (0, n)$. With this recursive formula, we are now equipped to solve the problem. We consider the subtasks with increasing difficulty.

Burning coins solution (20 points). *The restrictions in the first test set turn this problem into an equivalent of the Burning coins problem. If you solved Burning coins, you could simply copy your solution for it, make some minimal modifications, and you would get 20 points.*

Since in the first test set we only have two players and $k = 1$ (that is, we are interested in the winnings of the second player), this makes the problem equivalent to Burning coins, except that in Burning coins we were interested in the winnings of the first player, and in the current scenario we are interested in the winnings of the second player.

Let $w(i, j)$ denote the maximal winnings for the second player if the available coins are the ones with values $x_i, \ldots, x_{j-1}$. Then we have the following recursive relation for $w(i, j)$, which comes directly from applying the recursive formulas above for two consecutive turns, one of the first player and one of the second player:

$$w(i,j) = \begin{cases} \min\big(\max\big(x_{i+1} + w(i+2,j), x_{j-1} + w(i+1,j-1)\big), \\ \qquad \max\big(x_i + w(i+1,j-1), x_{j-2} + w(i,j-2)\big)\big) & \text{if } j - i \geqslant 2, \\ 0 & \text{if } j - i < 2. \end{cases}$$

This is because we have to consider both possible options for the move of the first player—taking either the leftmost coin x_i or the rightmost coin x_{j-1}. For each of these options, we have to consider two possible moves of the second player and take the one that yields the best winnings for the second player. Out of the two possible moves for the first player, we take the one that gives worse winnings for the second player as we do not know how the first player will behave.

The maximum winnings that the second player is then guaranteed to get is given by $w(0, n)$. We can find it in $O(n^2)$ time using dynamic programming.

Three player game solution (60 points). The next subtask is similar to the previous one, only now we know that we can have up to three players, and the k-th player is always the player that plays last. We can solve this by considering all cases depending on how many players there are. For the case of $m = 1$, there is only one player and he/she will get all the coins, so the winnings of the k-th player are the sum of the values of all the coins. The case of $m = 2$ was solved in the previous subtask. The case of $m = 3$ remains. We can solve it analogously to the case $m = 2$, except that this time we have to think three moves ahead instead of two.

Let $w(i, j)$ again denote the maximal winnings for the last player if the available coins are the ones with values $x_i, \ldots, x_{j-1}$. Then we have the following recursive relation for $w(i, j)$:

$$w(i,j) = \begin{cases} \min\big(\max\big(x_{i+2} + w(i+3,j), x_{j-1} + w(i+2,j-1)\big), \\ \qquad \max\big(x_i + w(i+1,j-2), x_{j-3} + w(i,j-3)\big), & \text{if } j - i \geqslant 3, \\ \qquad \max\big(x_{i+1} + w(i+2,j-1), x_{j-2} + w(i+1,j-2)\big)\big) \\ 0 & \text{if } j - i < 3. \end{cases}$$

This is because we have to consider three possible options for the move of the first two players—they could both take the leftmost coin, they could both take the rightmost coin, or one of them could take the leftmost coin, and the other—the rightmost coin. For each of these options, we have to consider two possible moves of the third player and take the one that yields the best winnings for him/her. Out of the three possible scenarios for the behaviour of the first and the

second players, we take the one that gives worse winnings for the third player as we do not know how the other players will behave.

The maximum winnings that the k-th player is guaranteed to get is then given by $w(0, n)$. We can find it using recursion and memoization, just like in the $m = 2$ case. The implementation of the recursive function for the case of $m = 3$ follows. Note that a full solution for 60 points would include a different recursive function for the case of $m = 2$, which we do not provide here.

```
// x[i] is the value of the i-th coin, and
// dp[i][j], where i <= j is the maximum possible winnings
// of the last player given that the game starts with coins
// x[i], x[i+1], ..., x[j-1].
// This recursive function computes dp[i][j], that is, the maximum possible
// winnings of the last player given that the game starts with coins
// x[i], x[i+1], ..., x[j-1] and given that there are three players.
int max_winnings_3(int i, int j, const VI& x, VVI& dp) {
    // If we have already computed the value for this state,
    // we don't compute it again.
    if (dp[i][j] != -1) return dp[i][j];
    // If there are fewer than 3 coins available, the last
    // player will not even get a coin, so his/her winnings are 0.
    if (j - i < 3) return dp[i][j] = 0;

    int winnings_first_two_players_take_leftmost_coins = std::max(
        x[i+2] + max_winnings_3(i+3, j, x, dp),
        x[j-1] + max_winnings_3(i+2, j-1, x, dp));

    int winnings_first_two_players_take_rightmost_coins = std::max(
        x[i] + max_winnings_3(i+1, j-2, x, dp),
        x[j-3] + max_winnings_3(i, j-3, x, dp));

    int winnings_first_two_players_take_leftmost_and_rightmost_coin = std::max(
        x[i+1] + max_winnings_3(i+2,j-1, x, dp),
        x[j-2] + max_winnings_3(i+1,j-2, x, dp));

    return dp[i][j] = std::min(std::min(
        winnings_first_two_players_take_leftmost_coins,
        winnings_first_two_players_take_rightmost_coins),
        winnings_first_two_players_take_leftmost_and_rightmost_coin);
}
```

We again have $O(n^2)$ possible states, and the recursion is called with each state at most a constant number of times, which gives us a time complexity of $O(n^2)$. This solution works for the first two test cases, which yields 60 points.

m player game solution (100 points). *In order to get the full 100 points, we need to handle an arbitrary number of players and general k.*

We now extend our logic to the general case. Let us use $w(i, j)$ to denote the maximal winnings for the *first* player if the available coins are the ones with values $x_i, \ldots, x_{j-1}$. We now think m steps ahead, just like we were thinking 3 steps ahead before. The recurrent relation for $w(i, j)$

is thus:

$$w(i,j) = \begin{cases} \max\big(x_i + \min\limits_{h\in\{0,\ldots,m-1\}} w\big(i+h+1, j-(m-h-1)\big), \\ \qquad x_{j-1} + \min\limits_{h\in\{0,\ldots,m-1\}} w\big(i+h, j-(m-h)\big)\big) & \text{if } j-i \geqslant 1, \\ 0 & \text{if } j-i < 1. \end{cases}$$

Notice that here we allow $w(i,j)$ with $i > j$ (and define it to be 0) as it lets us handle border cases easily. The recurrent relation intuitively means the following: we have to consider two possible options for the move of the first player—he/she could either take the leftmost coin or the rightmost coin. For each of these options, we have to consider all possible moves of the rest of the players and take the one that yields the worst winnings for the first player, because we do not know how the other players will behave. Out of the two possible scenarios for the move of the first player, we take the one that gives better winnings for him/her, as he/she is trying to maximize his/her own winnings.

What is left is to take care of the fact that we are interested in the winnings of player p_k, and not the first player. To turn player p_k into the first player, we consider all options for the behaviour of the first k players, that is, the players before p_k, on their first turn. This allows us to express the quantity we are looking for (the maximum winnings of player p_k for the entire game) in terms of $w(i,j)$ for various values of i and j. Let us denote the maximum winnings of player p_k for the entire game by $\mathcal{M}$. Then we have

$$\mathcal{M} = \min_{h\in\{0,\ldots,k\}} w\big(h, n-(k-h)\big).$$

This holds because out of all possibilities for the moves of the first k players, we pick the one that gives player p_k the worst winnings. There are $k+1$ options for the moves of the first k players: it could be that all of them pick the rightmost coin, or one picks the left one and all the rest pick the rightmost coin, etc., or that all of them pick the leftmost coin.

We can compute $w(i,j)$ using dynamic programming. To analyse the complexity of this algorithm, notice that we only need to compute $w(i,j)$ for some states (i,j), namely the states such that it is player p_k's turn. As we discussed above, this implies that $(n-j+i) \mod m = k$. Since this is true for only a $\frac{1}{m}$ fraction of the states, this means we need to compute $w(i,j)$ for $O(\frac{n^2}{m})$ states. For each state, we incur $O(m)$ time complexity to compute the minimum in the recursive relation. This gives us our total time complexity of $O(\frac{n^2}{m} \times m) = O(n^2)$.

3 Implementation

The discussed algorithm can be implemented using either a bottom-up iterative dynamic programming approach, or its top-down recursive equivalent, as presented in the tutorial (see the slides from Week 2). Here we present the recursive version.

We keep the values of the coins in the vector `x[0],...,x[n-1]`. We define a recursive function `max_winnings`, which computes the maximum possible winnings of the first player, given that the game starts with coins $x_i,\ldots,x_{j-1}$. We use memoization when computing `max_winnings(i,j)`, storing it in `dp[i][j]` once it has been computed, thus never computing the value of the recursion for the same state more than once. We directly implement the recursive formula described above in the recursive function.

4 Appendix

Implementation of the m player game solution *The following program is a recursive implementation of the $O(n^2)$ algorithm described above using memoization.*

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

typedef std::vector<int> VI;
typedef std::vector<VI> VVI;

// global upper bound on winnings: <= 1000 coins of value <= 2^{10}
// (fits into a 32-bit int)
const int INF = 1000 * 1024 + 1;

// n = #coins, m = #players,
// x[i] is the value of the i-th coin, and
// dp[i][j], where i <= j is the maximum possible winnings
// of the first player given that the game starts with coins
// x[i], x[i+1], ..., x[j-1].
// This recursive function computes dp[i][j], that is, the maximum possible
// winnings of the first player given that the game starts with coins
// x[i], x[i+1], ..., x[j-1].
int max_winnings(int i, int j, int n, int m, const VI& x, VVI& dp) {
    // If (i, j) is not a valid interval, return 0.
    if (j < i) return 0;
    // If there are fewer than 1 coins available, the first
    // player will not get a coin, so his/her winnings are 0.
    if (j - i < 1) return dp[i][j] = 0;
    // If we have already computed the value for this state,
    // we don't compute it again.
    if (dp[i][j] != -1) return dp[i][j];

    // The following implements the recurrent formula from above.
    int winnings_kth_left = INF, winnings_kth_right = INF;
    for (int l = 0; l < m; ++l) {
      winnings_kth_left = std::min(
          winnings_kth_left,
          max_winnings(i + l + 1, j - (m - l - 1), n, m, x, dp));
      winnings_kth_right = std::min(
          winnings_kth_right,
          max_winnings(i + l, j - (m - l), n, m, x, dp));
    }
    dp[i][j] = std::max(x[i] + winnings_kth_left, x[j-1] + winnings_kth_right);
    return dp[i][j];
}

int main() {
    std::ios_base::sync_with_stdio(false);
    int t;
    // Read in the number of test cases.
    for (std::cin >> t; t > 0; --t) {
        int n, m, k;
        std::cin >> n >> m >> k;

        // Read in all the values of the coins.
        VI x;
        x.reserve(n);
```

```
        for (int i = 0; i < n; ++i) {
            int v;
            std::cin >> v;
            x.push_back(v);
        }

        // Make sure our dp array is initialized to -1, which stands for
        // "not computed yet".
        VVI dp(n+1, VI(n+1, -1));

        // Consider all options for the behaviour of the first `k` players.
        // Any number of them (0 through k) could take the leftmost coins.
        // We take the minimum winnings for the k-th player out of all these.
        int maximal_winnings = INF;
        for (int i = 0; i <= k; ++i)
            maximal_winnings = std::min(
                maximal_winnings,
                max_winnings(i, n - (k - i), n, m, x, dp));
        std::cout << maximal_winnings << "\n";
    }
}
```