It is convenient to know the shortest path distances (the minimum time needed to travel) between any two "special" locations on the grid: the players' base, the enemy base, and the locations of the treasures. It is optimal for players to move only between special locations and only through shortest paths. Any aimless wandering can only make the answer worse. Computing these distances can be done in $\mathcal{O}(nrc)$ by performing BFS from each of the $n+2$ special locations. After that, we can forget about the grid and reframe the problem in terms of a weighted graph where the $n+2$ special locations are the nodes and the distances between special locations are the edge weights.

1 Subtask 1

If there is only one treasure, there is only a small number of possible players' actions:

- Do nothing (get zero points)
- Take the treasure but do not attack the enemy base
- Attack the enemy base but do not take the treasure
- Take the treasure and then attack the enemy base
- Attack the enemy base and then take the treasure

For each of these possibilities, we can check if the actions can be done within T seconds. This is equivalent to checking if the minimum time needed to perform the actions is less than or equal to T. The final answer is the score of the maximum-scoring possibility among those that can be done within T seconds.

Now we need to compute the minimum time needed to perform the actions. Notice that if there is only one treasure, we can pretend there is only one player. Having all players make the exact same moves doesn't change the answer:

- It's pretty obvious that all players can make the exact same moves without changing the amount of time required to do any of the following: doing nothing, only taking the treasure, or only attacking the base.
- If the players take the treasure and then attack the enemy base, it doesn't matter if the players split up, with some players going to the enemy base first. Every one needs to wait for the player who takes the treasure to reach the enemy base anyway, so they might as well make the exact same moves.
- If the players attack the enemy base and then take the treasure, the players gain nothing by splitting up before attacking the enemy base. Only one player needs to take the treasure afterward, but the others can make the same moves as the one player without changing the time required.

Therefore, we only need to compute the times required when there is only one player. This can be easily done after the distances between special locations have been pre-computed.

2 Subtask 2

This can be solved similarly to Subtask 1, except that the set of possible players' actions is bigger, and we pretend there are at most two players instead of at most one. The at-most-two-players assumption can be justified the same way as in Subtask 1. Taking only one treasure or attacking the enemy base in between taking the two treasures does not require the players to split up and can be handled exactly the same way as in Subtask 1. The interesting cases left are when both treasures are taken before or both after attacking the enemy base, or without attacking the enemy base. In these cases, it is optimal for the two players to split up when taking the treasures. If they do not attack the enemy base, they can just take independent paths and the time required is the greater one between the times required for the two paths. It turns out we can do the same thing for the other cases:

- If they attack the base before taking the treasures, making each player take shortest paths means they'll reach the base at the same time, so there is no need to do anything special to coordinate the time of the attack.

- If they take the treasures before attacking the base, the player who reaches the base earlier will just wait for the other one. The time required is still just the greater one between the times required for the two paths.

3 Subtasks 3 to 5

If there is only one player, there is no need to coordinate the enemy base attack. The enemy base works exactly like a treasure. The problem can then be reduced to the Shortest Hamiltonian Path Problem (SHPP). You can learn about this problem and its solution by first trying to solve AtCoder Beginner Contest 190E - Magical Ornament, consulting the editorial, and reading the relevant chapter from the Competitive Programmer's Handbook.

Since n is small, we can try every subset of treasures, for each one solving SHPP on only the corresponding subset of nodes, and taking the maximum-scoring one among those for which the SHP distance is less than or equal to T. Subtasks 3 to 5 correspond to different ways of solving SHPP, from slowest to fastest:

- (Subtask 3, $O(2^n n! n)$ time) For each permutation P of the n nodes, compute the time required to visit the n nodes in the order defined by P. Take the minimum among all the times.
- (Subtask 4, $O(n^2 2^n T)$ time) Use dynamic programming with the following function: $\text{CAN}(S, v, t)$ is true if and only if it is possible to visit all nodes in S starting at v within time t.
- (Subtask 5, $O(n^2 2^n)$ time) Use dynamic programming with the following function: $\text{OPT}(S, v)$ is the minimum time required to visit all nodes in S starting at v.

4 Subtask 6

First, consider the case where the players do not attack the enemy base. If there are two players, some treasures will be taken by the first player, some taken by the second player, and some not taken at all. We can try all 3^n ways of assigning treasures to players. The time required to achieve a certain way is just the maximum between the answers to the two SHPP instances generated by the assignment. The two players can move independently so the SHPP instances can be solved independently. The final answer is just the score of the maximum-scoring way among those achievable within T seconds. Notice that the subproblems of the first player are exactly the same as those of the second player. So there are $O(2^n)$ possible SHPP instances and the answers to these can be shared among the 3^n ways by pre-computing the SHP distances or memoizing. The computation takes $O(n^2 2^n + 3^n n)$ time, $O(n^2 2^n)$ for solving all the SHPP instances with dynamic programming, and $O(3^n n)$ for trying all ways to assign the treasures to players and computing scores. Although not required, this last term can be improved to $O(2^n n + 3^n)$.

If the players need to attack the enemy base, treasures can belong to one of five groups:

1. taken by the first player before attacking the enemy base
2. taken by the second player before attacking the enemy base
3. taken by the first player after attacking the enemy base
4. taken by the second player after attacking the enemy base
5. not taken at all

For the first two groups, notice how it is optimal for a player to take the treasures and then go to the enemy base as fast as he can no matter what the other player does. One of the players can always just wait at the enemy base if necessary. After the attack, a player should again take the treasures as fast as possible no matter what the other player does.

Therefore, a particular assignment of treasures to groups produces four independent SHPP instances (nothing needs to be done for the fifth group), the time required for a particular assignment being the maximum of the first two

SHP distances plus the maximum of the last two SHP distances. Modify the first two SHPP instances slightly to require ending at the enemy base, and the last two to require starting at the enemy base. The first two share subproblems and the last two share subproblems, so only two independent DP memos are required.

Trying all 5^n assignments of treasures to groups is too slow though. Instead, notice that the treasures taken before the attack can be split between the two players independently of how the treasures taken after the attack are split between them. In other words, we can first pre-compute all 3^n times required for two players to take treasures and end at the enemy base, then independently pre-compute all 3^n times required for two players to start at the enemy base and take treasures, then try all 3^n ways to assign the treasures into these three groups:

1. taken before attacking the enemy base
2. taken after attacking the enemy base
3. not taken at all

Then the time required for a particular assignment into these three groups can be computed in $O(1)$ using the pre-computed values. The SHPP pre-computation takes $O(n^2 2^n + 3^n)$ time, $O(n^2 2^n)$ for solving all the SHPP instances with dynamic programming, and $O(3^n)$ for trying all ways to assign treasures to players. Then there is an additional $O(3^n n)$ or $O(2^n n + 3^n)$ time for trying all ways to assign the treasures to before/after/none and computing scores.

5 Subtask 7

We can generalize the solution to subtask 6 by trying all $O((x+1)^n)$ ways to assign the treasures to players (separate from trying all 3^n ways to assign the treasures to be taken before or after the attack). However, this is too slow.

Instead, notice that splitting a set of S treasures into X disjoint subsets can be expressed recursively: choose a subset S' of the S treasures to put in the Xth set and then recursively split $S-S'$ into $X-1$ disjoint subsets. This leads to the following dynamic programming solution: let $\text{OPT}(S, X)$ be the minimum time required to take all the treasures in S when there are X players. Then (omitting base cases) $\text{OPT}(S, X) = \min_{S' \subseteq S} \max(\text{OPT}(S - S', X - 1), \text{SHP}(S'))$, where SHP(S) is the minimum time required to visit all nodes in S. This can be easily modified to force all players to start at the players' base and end anywhere, to start at the players' base and end at the enemy base, or to start at the enemy base and end anywhere, covering all the cases needed to take attacking the enemy base into account.

Implementing the resulting recurrences with a straightforward DP takes $O(4^n x)$ time. This can be improved to $O(3^n x)$ by properly skipping subsets S' which are not subsets of the currently considered S in the DP transition step, as illustrated in the "Suboptimal Solution" section of this SOS Dynamic Programming Tutorial on Codeforces. The total running time of the intended solution is $O(nrc + n^2 2^n + 3^n x)$.

6 Bonus

It is possible to solve this problem for an arbitrarily large number of players.