The city of UNSW has decided to partition High Street into voting districts. There are n houses on High Street, living on consecutive addresses labelled $1, 2, \dots, n$. Each voting district is an interval of addresses $i, i+1, \dots, j$ with $1 \le i < j \le n$. The city has enforced two constraints:

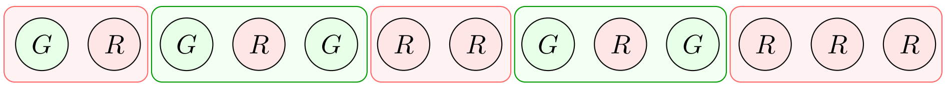
(1) Each house on High Street must belong to *exactly* one voting district.

(2) The number of houses in each district must lie between k and $2k$, where k is a parameter.

In each election, houses can vote for either Oceania or Eurasia but they can only vote for one party. Since the majority of houses in the city of UNSW support Oceania, the city deems a district *favourable* if more than half of the houses in the district support Oceania; otherwise, the district is *unfavourable*. Of course, the city has a complete record of all of the votes in the election.

The next election is coming up and the city has decided to ask you, as the budding algorithmist, to help assign districts. You are given a boolean array $\textsc{Vote}[1..n]$, where $\textsc{Vote}[i] = 1$ if resident i voted for Oceania and $\textsc{Vote}[i] = 0$ if resident i voted for Eurasia. You are also given an integer k. Describe an $O(nk^2)$ algorithm that returns the largest number of favourable districts in any legal partition of the houses.

For example, consider the following legal partition of $n = 13$ houses with $k = 2$ (therefore, each district can be assigned 2 to 4 houses). Houses who voted for Oceania are coloured green and the houses who voted for Eurasia are coloured red. There are two favourable districts (coloured green) and two unfavourable districts (coloured red).



A legal partition of houses to five districts. The first district is unfavourable, the second district is favourable, the third district is unfavourable, the fourth district is favourable, and the fifth district is unfavourable.

Note. *You may assume that a legal partition always exists. There is also an $O(kn)$ algorithm by doing some precomputation.*

Solution. We begin by initialising a cumulative array of the votes, allowing us to calculate the total votes between two houses efficiently. We will call this PS with $PS[i] = PS[i-1] + \textsc{Vote}[i-1]$. We now define a Dynamic Programming table DP, all initialised to 0. Our first subgroup must be no smaller than k, so we will iterate through each subsequence ending at $i = k$ to $i = n$. For each i, we check each possible size, j, randing from k to i (or $2k$ where possible). If it is such that our division has a majority favourable vote, we update $DP[i] = \max\{DP[i], DP[i-j] + 1\}$. In other words, we look at the best result we could achieve before this group started and see if adding this new group would give us a better result than what we currently have for position i. Our maximum size will be the largest element in DP.

Since we check k cases for each group ending position i, we would have a time complexity of $O(kn)$.