W – solution

Author: Zoltán Szabó

Recurrence

By definition, a W-shaped array must contain one local maximum and two local minima situated to the left and right of the local maximum. If any of these extremes are repeated, we refer to all their occurrences collectively. For example, in the array

$V = (5, 4, 2, 2, 2, 3, 5, 7, 7, 7, 6, 6, 3, 8)$

the local maximum is (7, 7, 7) and the local minima are (2, 2, 2) and (3). Let us divide the array into 5 non-overlapping zones as shown in Figure 1.

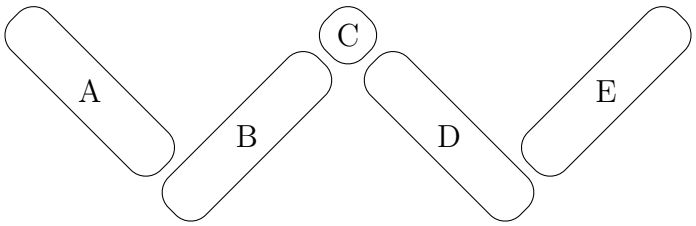

Figure 1: Breakdown of a W.

- zone A contains all the elements before the first local minimum;
- zone B contains the elements from the first local minimum up to, but not including, the local maximum;
- zone C contains just the local maximum;
- zone D contains the elements after the local maximum, up to and including the last local minimum;
- zone E contains the elements after the last local minimum.

In the example above, $A = (5, 4)$, $B = (2, 2, 2, 3, 5)$, $C = (7, 7, 7)$, $D = (6, 6, 3)$, $E = (8)$. From the requirement that every segment contain two distinct values, it follows that all 5 zones must be non-empty if the array is W-shaped.

We now sort the array and process every group of equal values, from the smallest to the largest, according to the following rules:

1. The group may be placed in one zone or distributed among several zones.
2. All zones must receive at least one element.
3. Zone A may receive elements only after zone B received at least one element. Similarly, zone E may receive elements only after zone D received at least one element.
4. Zone C may receive elements only after zones B and D received at least one element each.
5. Once zone C received its elements, zones B, C and D may not receive any more elements.

We use five bits to keep track of empty (0) and occupied (1) zones. Thus the 5-bit state 01000 means that so far we have been putting elements in zone B only. Naturally, we start in state 00000 and must count the number of ways to process all groups and end up in state 11111. Therefore, we are looking for a recurrence relation of the form

$$C_s(k) = \sum_{s'} C_{s'}(k-1) \cdot W(s', s, G(k))$$

where

- k is the index of the group of equal values we are currently distributing;
- s is the 5-bit state after distributing group k;
- s' are the possible states before distributing group k;
- $G(k)$ is the number of equal elements in group k;
- $W(s', s, G(k))$ is the number of ways of distributing $G(k)$ equal elements in state s' so as to end up in state s.

We can compute all these values by hand, but the process is tedious and error-prone. A safer approach is to compute them programmatically. We loop through every pair of 5-bit states and determine if (a) the states are valid and (b) the transition between them is valid. Although theoretically there are $2^5 = 32$ states, many of them are invalid. For example, state 00001 is invalid because zone E cannot have elements while zone D is empty (rule 3). Furthermore, some transitions are invalid: we cannot transition from

01000 to 01110 using a single group of values, because we need to assign elements to zone D before assigning (larger) elements to zone C (rule 4). We are generally allowed to loop to the same state: going from 01010 to 01010 means we keep adding elements to zones B and D only. The complete state diagram is shown in Figure 2.

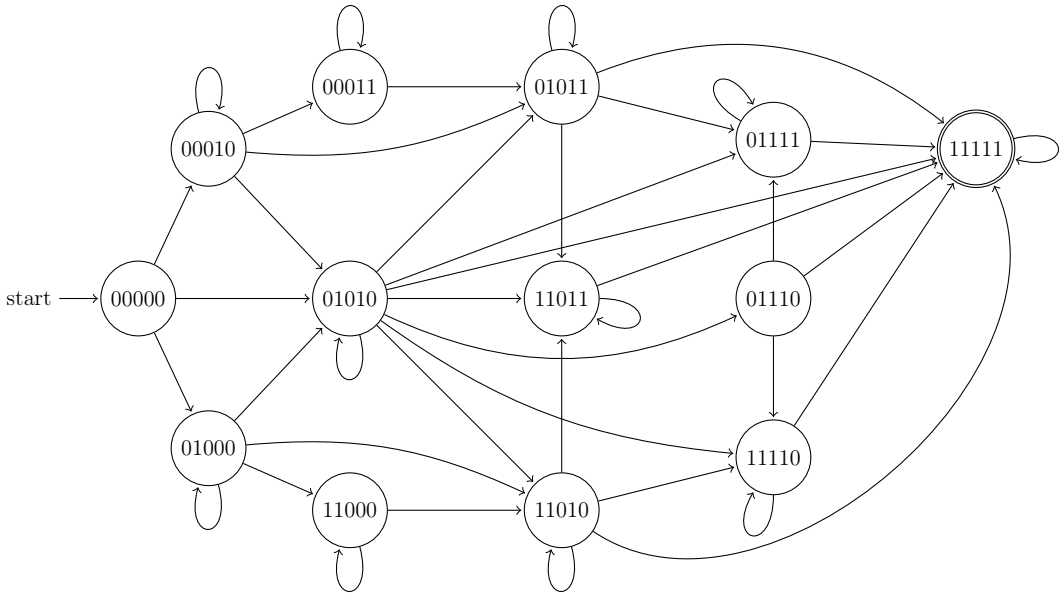

Figure 2: State diagram for zone transitions.

What about the W coefficients? For example, how many ways are there to distribute $G(k) = 10$ equal values while moving from state $s' = 01000$ to state $s = 11010$? Clearly we must distribute one element each to zones A and D, as those zones are newly occupied during this transition. The remaining eight values can go to any of the three zones A, B and D. The formula for this, which we provide without proof, is known as *combinations with repetition*:

$$\left(\!\!\binom{8}{3}\!\!\right) = \binom{8+3-1}{3-1} = 45$$

In general, if s has $bits$ bits of 1, of which new are new in comparison to s', then the formula is:

$$W(s', s, G(k)) = \left(\!\!\binom{G(k) - new}{bits}\!\!\right) = \binom{G(k) - new + bits - 1}{bits - 1}$$

It follows that we simply need to count the new and total bits for every state transition. These values need to be adjusted when zone C is involved. When zone C is occupied for the first (and only) time, zones B and D may not be modified at the same time, so we

subtract 2 from *bits*. Similarly, when zone C is already present in s', we may not modify any of the zones B, C or D, so we subtract 3 from *bits*.

Optimizations

Since computing $C_s(k)$ only depends on $C_s(k-1)$, we do not need to store an $O(n)$ matrix of data, but only two rows. Once $C_s(k)$ is computed for all s, we can discard and reuse the space for $C_s(k-1)$.

Symmetrical states will lead to equal values, which means that we can keep one copy and count it twice in transitions. For example, state 01000 is symmetrical to 00010 and they both participate in $C_{01010}(k)$. Therefore, we discard 01000 completely and count 00010 twice when computing $C_{01010}(k)$. This is sensibly faster, although not required for a perfect score.

The number of zones where we distribute values from a single group can be at most 4. It can never be 5, since if zone C is occupied, it prevents us from using zones B, C and D ever again. Therefore $bits - 1 \leq 3$ and we can compute combinations on the fly using multiplications and divisions. Precomputing the combinations reduces the running time by some 10-20%, but this is not required for a perfect score.

Special case: two distinct values

Let the values be $v_1 < v_2$, having c_1 and c_2 occurrences respectively. Consider the string of all the occurrences of v_2 with $c_2 - 1$ gaps in between. To obtain the W shape, we must insert the value v_1 in two of the gaps. We can choose the gaps in ($c_2 - 1$ choose 2) ways. For a fixed pair of gaps, we can distribute the c_1 occurrences of v_1 in $c_1 - 1$ ways. Thus, the number of solutions is

$$S(c_1, c_2) = \frac{(c_1 - 1)(c_2 - 1)(c_2 - 2)}{2}$$

Special case: all distinct values (first method)

This method leads to a direct formula, which is a bit lengthy to calculate, but requires only basic math. When all the values are distinct, the only input value we need is n. Without loss of generality, let us assume that the element values are $-1, 0, 1, \ldots, n-2$. Clearly, the value -1 must be a local minimum. Let us assume it is the left one and we will double our answer to account for mirror solutions. Let m be the right local minimum

and p be the local maximum, with $0 \leq m < p \leq n-2$. Finally, let S_1, S_2, S_3 and S_4 be the four segments of the W from left to right, as shown in Figure 3.

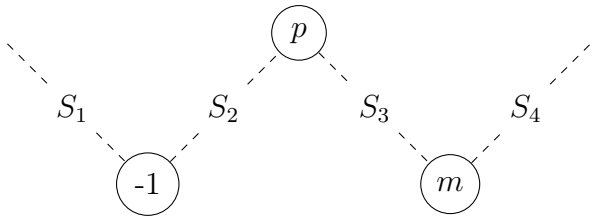

Figure 3: Construction of a W

Then, in order to satisfy the inequalities on every segment, we can distribute the other elements among segments as follows:

values (inclusive)	**segments**
$0 \dots m-1$	S_1, S_2
$m+1 \dots p-1$	S_1, S_2, S_3, S_4
$m+1 \dots n-2$	S_1, S_4

This distribution will count some illegal configurations, as S_1 and/or S_4 may remain empty (note that S_2 and S_3 are guaranteed to have at least two values). We therefore have to subtract the number of configurations that leave **either** S_1 or S_4 empty, then add back the number of configurations that leave **both** S_1 and S_4 empty, which we subtracted twice. The full table of possibilities is:

values	**segments**			
	regular	S_1 **empty**	S_4 **empty**	S_1, S_4 **empty**
$0 \dots m-1$	S_1, S_2	S_2	S_1, S_2	S_2
$m+1 \dots p-1$	S_1, S_2, S_3, S_4	S_2, S_3, S_4	S_1, S_2, S_3	S_2, S_3
$m+1 \dots n-2$	S_1, S_4	S_4	S_1	–

The rest of the algorithm just entails getting the formulas right. First, some notations:

- $S(n, p, m)$ denotes the number of regular distributions for a fixed p and m;
- $S'(n, p)$ denotes the number of regular distributions for a fixed p and all m;
- $S''(n)$ denotes the number of regular distributions for all p and m.

$$\begin{aligned} S(n,p,m) &= 2 \cdot 2^m \cdot 4^{p-m-1} \cdot 2^{n-p-2} \\ &= 2^{1+m+2\cdot(p-m-1)+n-p-2} \\ &= 2^{n-3} \cdot 2^{p-m} \end{aligned}$$

$$\begin{aligned} S'(n,p) &= \sum_{m=0}^{p-1} S(n,p,m) \\ &= 2^{n-3} \cdot (2^p + 2^{p-1} + \cdots + 2) \\ &= 2^{n-3} \cdot (2^{p+1} - 2) \\ &= 2^{n+p-2} - 2^{n-2} \end{aligned}$$

$$\begin{aligned} S''(n) &= \sum_{p=1}^{n-2} S'(n,p) \\ &= 2^{n-2} \cdot (2 + 2^2 + \cdots + 2^{n-2}) - (n-2) \cdot 2^{n-2} \\ &= 2^{n-2} \cdot (2^{n-1} - 2) - (n-2) \cdot 2^{n-2} \\ &= 2^{2n-3} - n \cdot 2^{n-2} \end{aligned}$$

We similarly define P, P', P'' and Q, Q', Q'' for the cases where S_1 and S_4 remain empty, which yields

$$P''(n) = \frac{3^{n-1}+1}{2} - n$$

$$Q''(n) = 3^{n-1} - 2^n + 1$$

For the last case, note that S_1 and S_4 can both be empty when $p = n-2$, otherwise we would have no place for elements larger than p. We get

$$R''(n) = 2^{n-1} - 2$$

and the answer is

$$\begin{aligned} A(n) &= S''(n) + P''(n) + Q''(n) + R''(n) \\ &= 2^{2n-3} - n \cdot 2^{n-2} - \frac{3^{n-1}+1}{2} + n - 3^{n-1} + 2^n - 1 + 2^{n-1} - 2 \\ &= 2^{2n-3} - n \cdot 2^{n-2} - \frac{3^n+7}{2} + n + 3 \cdot 2^{n-1} \end{aligned}$$

Special case: all distinct values (second method)

Let A, B, C, D and E be the five elements in the corners of the W, in increasing order. There are only 16 valid ways to arrange these elements in the corners, which can be determined experimentally. For example, A, being the minimum, must be in the lower corners. Furthermore, every two arrangements are mirror images, such as $CBDAE$ and $EADBC$, shown in Figure 4.

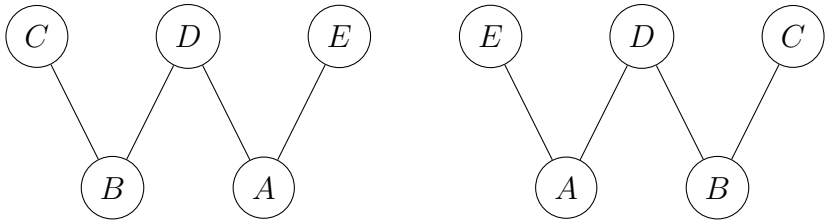

Figure 4: Corner values and symmetry

Now let X, Y, Z and T be the distances between these five elements, as shown in Figure 5.

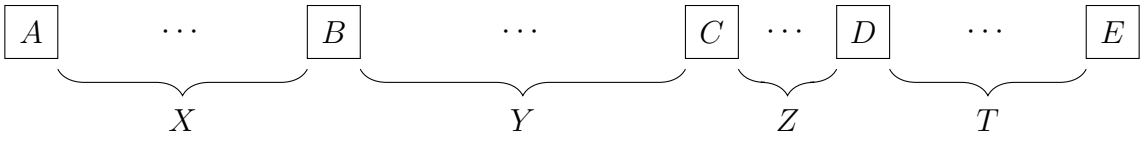

Figure 5: Array spacing between corner elements

For the case $CBDAE$, where can the X elements go in the permutation so as to respect the W ordering? On the segments DA and AE, so there are 2^X ways of distributing these elements. Similarly, the Z elements can go on the segments BD, DA and AE, so there are 3^Z ways of distributing them.

In general, for every corner arrangement and for every quadruple (X, Y, Z, T) we need to compute $a^X \cdot b^Y \cdot c^Z \cdot d^T$, where a, b, c, d depend on the corner arrangement and can be computed by hand with relative ease. The answer is the sum of all products. The question is, how do we compute it in $O(n)$?

For a fixed C (so fixed $X + Y$), the number of ways to distribute elements on its left (between A and C) is

$$S = a^{X+Y} \cdot b^0 + a^{X+Y-1} \cdot b^1 + \cdots + a^0 \cdot b^{X+Y}$$

When moving C one position to the right, the number of ways becomes

$$\begin{aligned} S' &= a^{X+Y+1} \cdot b^0 + a^{X+Y} \cdot b^1 + ... + a^0 \cdot b^{X+Y+1} \\ &= S \cdot a + b^{X+Y+1} \end{aligned}$$

It follows that these values can be calculated in $O(1)$ from one another. The base case is $C = 2$, when clearly $A = 0$, $B = 1$, $X = Y = 0$ (no elements between A, B and C) and $S = 1$.

The same applies to C, D, E, Z, T, c and d to the right of C. The answer (for an arbitrary corner arrangement) is the convolution of the values obtained on either side of C.

Naive approach

We could simply generate all the distinct permutations and count those that are W-shaped. This is obviously inefficient and should only score 15 points. However, it works reasonably fast for small test cases and it should only take 15-20 minutes to implement. It can be useful in double-checking and debugging the efficient implementations, which are much more tedious and bug-prone.

To achieve this, we can backtrack at every position through all the distinct available values. To avoid duplicates at every step, we keep a bit mask of used values. This is cheap, although it limits the naive approach to values between 1 and 31.

As for checking the W-shape, it is best to do this during the backtracking and prune early, rather than wait until each permutation is generated. This can be done a number of ways. We consider the shortest and most elegant to involve the finite state automaton in Figure 6 to track our progress along the W.

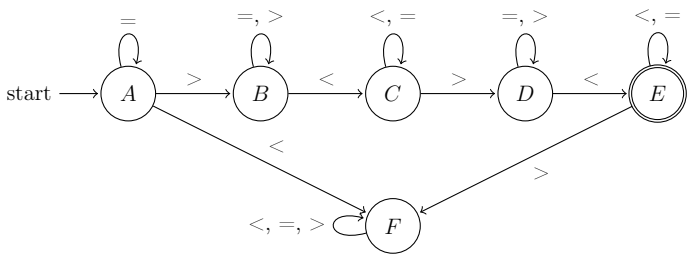

Figure 6: Finite state automaton for checking the validity of a W-shape.

Its states are:

state	meaning
A (start)	on segment 1; possibly read some equal values, but no inequality yet
B	on segment 1; read a smaller value; safe to move on to segment 2
C	on segment 2; read a larger value; safe to move on to segment 3
D	on segment 3; read a smaller value; safe to move on to segment 4
E (accepting)	on segment 4; safe to end at any time
F (error)	array is not W-shaped; absorbing state

The transitions between states are not based on array values, but on comparisons between consecutive values. For example, "<" means "the previous value was smaller than the current value". This translates to the trivial code:

```
int STATES[6][3] =
  {
   { 5, 0, 1 }, /* 0 = state A */
   { 2, 1, 1 }, /* 1 = state B */
   { 2, 2, 3 }, /* 2 = state C */
   { 4, 3, 3 }, /* 3 = state D */
   { 4, 4, 5 }, /* 4 = state E */
   { 5, 5, 5 }, /* 5 = state F */
  };

void bkt(int level, int state) {
  if (state == ERROR_STATE) {
    return;
  }

  if (level > n) {
    if (state == ACCEPTING_STATE) {
      numSolutions++;
    }
    return;
```

```
  }

  for (int i = level; i <= n; i++) {
    ...
    int cmp = CMP(v[level - 1], v[level]); /* should return 0, 1 or 2 */
    bkt(level + 1, STATES[state][cmp]);
    ...
  }
}
```