Code Jam 2014 - World Finals

Analysis: Power Swapper

Video of Bartholomew Furrowâ€™s explanation.

Observe that a swap operation of size 2^x can be performed independently from a swap operation of size 2^y. Thus, we can simplify the sorting process by performing the swaps from the smallest size to the largest size and assume that when performing a swap of size 2^z, the adjacent numbers inside it are **fully** sorted (by the previous, smaller sized swaps). **Fully** sorted here means the adjacent numbers are increasing consecutively (i.e., each number is bigger than the previous by 1).

Letâ€™s look at the possible swappings for the smallest 2^k size element where k = 0 (i.e., swapping two sets of size 1). There are 2^N valid sets of size 1. However, we are only interested in swapping two valid sets such that all 2^(k+1) sized valid sets are fully sorted. This is important because when swapping larger sizes, we assume the smaller sized valid sets are all already fully sorted. Letâ€™s look at some examples:

- Permutation: 2 4 1 3. The only way is to swap 2 with 3, yielding 3 4 1 2 where all 2^1 sized valid sets are fully sorted.
- Permutation: 1 4 3 2. There are two ways to perform a swap so that all 2^1 sized valid sets are fully sorted. The first is by swapping 1 with 3 and the other is by swapping 4 with 2.
- Permutation: 1 4 3 2 6 5. In this case, there are 3 valid sets of size 2^1. It needs at least two swaps to make all 2^1 sized valid sets fully sorted. Thus, it is impossible to sort this permutation.

Learning from the examples above, it is sufficient to only consider swapping at most two valid sets of size 2^k for k = 0. We can generalize this for larger k, assuming all valid sets of smaller size are already fully sorted.

To count the number of ways to sort, we can use recursion (backtracking) to simulate all possible swaps. The recursion state contains the underlying array, the value k, and the number of swaps performed so far. The recursion starts from the smallest sized swaps (k = 0) with the original input array, then it decides which valid sets to swap (if any) and recurses to the larger (k + 1) swaps, and so on until it reaches the largest sized swaps (k = N). The depth of the recursion is at most N and there are at most two branches per recursion state (since there are at most two possible swaps for each size) and each state needs O(2^N) to gather at most two valid candidate sets to be swapped. Thus, the algorithm runs in O(2^(N*2)). When the recursion reaches depth N, it has the number of swaps that have been done so far. Since the ordering of the swaps matters, the number of ways is equal to the factorial of the number of swaps. We propagate the number of ways back up to the root to get the final value. Below is a sample implementation in Python 3:

```
import math

def swap(arr, i, j, sz):  # Swap two sets.
  for k in range(sz):
    t = arr[i + k]
    arr[i + k] = arr[j + k]
    arr[j + k] = t

def count(arr, N, k, nswapped):
  if k == N:
    return math.factorial(nswapped)
```

```
  i = 0
  idx = []  # Candidatesâ€™ index for swappings.
  sz = 2**k
  while i < 2**N:
    if arr[i] + sz != arr[i + sz]:
      idx.append(i)
    i = i + sz * 2

  ret = 0
  if len(idx) == 0:  # No swap needed.
    ret = count(arr, N, k + 1, nswapped)

  elif len(idx) == 1:  # Only one choice to swap.
    swap(arr, idx[0], idx[0] + sz, sz)
    if arr[idx[0]] + sz == arr[idx[0] + sz]:
      ret = count(arr, N, k + 1, nswapped + 1)
    swap(arr, idx[0], idx[0] + sz, sz)

  elif len(idx) == 2:  # At most 2 choices.
    for i in [idx[0], idx[0] + sz]:
      for j in [idx[1], idx[1] + sz]:
        swap(arr, i, j, sz)
        if arr[idx[0]] + sz == arr[idx[0] + sz]:
          if arr[idx[1]] + sz == arr[idx[1] + sz]:
            ret = ret + count(arr, N, k + 1, nswapped + 1)
        swap(arr, i, j, sz)

  return ret


for tc in range(int(input())):
  N = int(input())
  arr = list(map(int, input().split()))
  print("Case #%d: %d" % (tc+1, count(arr, N, 0, 0)))
```