Blind Sort

Christopher He

June 15, 2022

1 Introduction

We are given an array of integers to sort but we are not allowed to see the contents of the array. The only operation we have is to swap any two elements. This swap operation will return if we placed any of those two elements in its correct sorted position. Once an element is in its correct sorted position, it will "freeze" and will no longer move. Originally, all elements are out of place. We will explore a probabilistic algorithm that sorts the array by randomly swapping elements and prove it is expected to sort in $\Theta(N^2)$ swaps. We will also explore a deterministic algorithm to sort the array that also runs in $\Theta(N^2)$ time. Lastly, we will provide the model of an adversary that would force any algorithm to make $\Omega(N^2)$ swaps to sort the array, thus establishing a lower bound.

In studying this problem, we will see that we are really working with derangements. A derangement is a permutation of elements such that no element is in its correct position. For our problem, assume a derangement of size N is an array A of integers {1, 2, ... N} such that $A[i] \neq i$ for all $i = 1, ..., N$.

We also introduce the concept of a good swap. A good swap is a swap that places at least one element in its correct position. That is, a swap (i, j) is good if $A[i] = j$ or $A[j] = i$. Call a good swap that freezes exactly one element a 1-swap and a good swap that freezes two elements a 2-swap.

We familiarize the reader to the structures we are working with by considering a counting problem: how many derangements of size N are there with K good swaps?

1.1 Number of derangements with k good swaps

Given a permutation, we can find the cycle decomposition of it. For example, given $A = [3, 2, 1, 6, 4, 7, 5]$ the cycle decomposition is $[3, 1], [2], [6, 7, 5, 4]]$. We first make some observations about the cycle decomposition of a derangement. First, there would be no cycles of length 1, as this would represent an element in its original position. Also, a cycle of length 2 would represent a 2-swap. Lastly, a cycle of length l where $l > 2$ would represent l 1-swaps. Given these observations, let c_2 be the number of cycles of length 2 (2-cycle) in a derangement, then the number of good swaps is $N - c_2$.

If we find the expected number of good swaps, this could also give us insight into the expected number of 2-cycles. Note we could've found the expected number of 2-cycles directly, but we saw the relation after the fact.

Let s be the number of good swaps in a derangement D of size N. We are trying to find

$$E[s] = \sum_{k=\lfloor N/2 \rfloor}^{N} p_k \times k$$

Where p_k is the probability of a derangement having k good swaps. We can define this as

$$\frac{\text{number of derangements of size } N \text{ with } k \text{ good swaps}}{\text{total number of derangements of size } N}$$

It can be shown that the number of derangements of size N is exactly $\lfloor \frac{N!+1}{e} \rfloor$. But how many derangements of size N have k good swaps? We propose a recurrence to answer this question.

We start with a naive recurrence: Let T(N, k) be the number of derangements of size N with k good swaps where $k <= N$. We define a recurrence for T(N, k) based on the following idea:

Pick i elements from the N elements, where $i > 1$. With these i elements, we will make a cycle of length i.

If we make a cycle of length 2, that contributes 1 good swap. We would like the rest of the $N - 2$ vertices to contribute $k - 1$ good swaps. Assume we have $T(N - 2, k - 1)$

If we make a cycle of length i where $i > 2$, that contributes i good swaps. Assume we have $T(N - i, k - i)$.

Let C_i be the number of cycles we can make of length i. There are $\binom{N}{i}$ ways to pick i vertices. With these i vertices, there are $(i - 1)!$ cycles we can make (fix a starting element, there are $(i - 1)$ choices for the second vertex in the cycle, then $(i - 2)$, and so on).

So $C_i = \binom{N}{i} \times (i - 1)!$.

Putting this all together, we have the following recurrence:

$$T(N, k) = C_2 \times T(N - 2, k - 1) + \sum_{i=2}^{k} C_i \times T(N - i, k - i)$$

This recurrence is naive because of the following observation: Take for example a derangement of 11 elements. Let's say we split this derangements into cycles of lengths 4, 3, 2, and 2. We counted this arrangement 4! times since there are that many ways to permute these cycles. Thus we define another recurrence $T'(N, k, s)$, the number of ways to make a derangement of N vertices with k good swaps with s cycles. The naive reccurence is slightly modified:

$$T'(N, k, s) = C_2 \times T'(N - 2, k - 1, s - 1) + \sum_{i=2}^{k} C_i \times T'(N - i, k - i, s - 1)$$

Thus our final solution is

$$T(N,K) = \sum_{s=1}^{N} \frac{T'(N,K,s)}{s!}$$

Running this algorithm shows that $E[S] = N - \frac{1}{2}$, which reveals that $E[c_2] = \frac{1}{2}$.

2 Deterministic Algorithm

We propose a simple deterministic algorithm: starting at index $i = 0$, we repeatedly swap the element at index i with all indices $j > i$ in a linear fashion, until we freeze index i in which case we go onto the next iteration of i. It is easy to see that this runs in $O(N^2)$ time. In fact, for any N we can construct an input that takes $\Theta(N^2)$ swaps. Let $N = 8$, for example, and consider the input $[5, 6, 7, 8, 4, 3, 2, 1]$. We can characterize this as putting the half of biggest elements in the beginning of the array, and the half of smallest elements at the end of the array in reverse order. In essence, we are trying to maximize the number of swaps to place the smallest elements where they belong. The first iteration will take $N - 1 = 7$ swaps until 1 is placed in its correct position. Observe that all elements between 5 and 1 are circularly shifted right by 1 in the process. So we have

$[5, 6, 7, 8, 4, 3, 2, 1] \rightarrow 7$ swaps
$[1, 5, 6, 7, 8, 4, 3, 2] \rightarrow 6$ swaps
$[1, 2, 5, 6, 7, 8, 4, 3] \rightarrow 5$ swaps
$[1, 2, 3, 5, 6, 7, 8, 4] \rightarrow 4$ swaps
$[1, 2, 3, 4, 5, 6, 7, 8]$

And the array is sorted. In general, each iteration i performs $N - i$ swaps. For an even N, the number of iterations is $N/2$. So, the total number of swaps is $(N-1) + ... + \frac{N}{2}$. We can describe this as a difference of two sums and solve:

$$
\begin{aligned}
&\sum_{i=1}^{N-1} i - \sum_{i=1}^{\frac{N}{2}-1} i \qquad (1)\\
&= \frac{N(N-1)}{2} - \frac{(\frac{N}{2})(\frac{N}{2}-1)}{2}\\
&= \frac{N(N-1)}{2} - \frac{(2N)(\frac{N}{2}-1)}{8}\\
&= \frac{N^2-N}{2} - \frac{N^2-2N}{8}\\
&= \frac{4N^2-4N}{8} - \frac{N^2-2N}{8}\\
&= \frac{3N^2-2N}{8} = \frac{N(3N-2)}{8}
\end{aligned}
$$

This attempt at a worst case input will be seen again when we describe the adversary for any algorithm. It turns out that when the deterministic algorithm "plays" the adversary, the adversary will force the algorithm to take this exact amount of steps (and claims this was the input all along).

It is interesting to note that the worst case does not always follow this structure. For $N = 10$ there is another derangement that make the deterministic algorithm take 1 more swap than for the derangement we constructed, namely $[6, 7, 8, 9, 3, 4, 5, 2, 1, 0]$. We spent some time trying to find a way to generalize this structure. One can revisit this by looking at the cycle decomposition of this worst case derangement.

Before, we observed that each iteration of the algorithm would circularly shift a subarray of elements. Because of this, some elements can be unintentionally frozen. This led to some roadblocks in finding a nice recurrence for the running time of the deterministic algorithm. Although, because the change to the array can be characterized nicely (circular shift to some elements), we did spend some time trying to find how many derangements are there such that k elements are in the index before their correct ones (so k elements would be shifted and frozen). If we found a nice answer to this, we could've described the running time as

$T(N) = \sum_{k=1}^{N} p_k * T(N-k)$

Although we were not able to find a recurrence for the running time, through simulations we found that the deterministic algorithm took around $N^2/6$ swaps.

2.1 Case where array has k distinct values

As the introduction mentioned, we have a lower bound of $\Omega(N^2)$ swaps on any blind sorting algorithm. In this section, we go over a case that would break our lower bound. In particular, if the array of length N only contains numbers from a set of K elements (for example $\{1, ..., K\}$) for $1 \leq K \leq N$, then we have a

deterministic $O(NK)$ algorithm as follows: we always maintain a pointer to the first nonfrozen index from the left, call it i. In each iteration, we swap i with all indices j such that $j > i$. At any time if index i is frozen, we increment i by 1. In each iteration, when we finish scanning all j, we repeat and our algorithm stops when $i > N$, that is, all elements are frozen. Notice that in each iteration i, all instances of the ith smallest element in K will be fixed. Thus, we will have at most K iterations, each performing $O(N)$ swaps so in total $O(NK)$ swaps. Note that for the general case $K = N$, this algorithm is still $O(N^2)$.

3 Probabilistic Algorithm

In the previous section we describe a deterministic algorithm to sort the array. We introduce a simple randomized algorithm: repeatedly swap random pairs of elements until the array is sorted (or all elements are frozen). It turns out that this algorithm runs in expected $\Theta(N^2)$ time.

3.1 Expected Number of Swaps is Between $N^2/2$ and $N^2/8$

As mentioned before, the number of good swaps in a derangement size N is $N - c_2$ where c_2 is the number of 2-cycles. Since c_2 must be between 0 and $N/2$, then the number of good swaps is between N and $N/2$. We consider the best case where there are N good swaps. Then the probability of a good swap is $\frac{N}{\binom{N}{2}} = \frac{2}{N-1}$. Given the probability of a good swap, we expect to perform $\frac{N-1}{2}$ swaps before we freeze at least one element. Since we are considering the best case, let's assume our good swap freezes two elements. Then we now have derangement of size $N - 2$ that, in the best case, has $N - 2$ good swaps. Thus we have the sum $\frac{1}{2}((N-1) + (N-3) + (N-5) + ... + 3 + 1) = \frac{1}{2}(\frac{N^2}{4}) = \frac{N^2}{8}$.

Similarly, let's consider the worst case where there are $N/2$ good swaps. Then the probability of a good swap is $\frac{1}{N-1}$ so we expect to perform $N - 1$ swaps before freezing something. Since we are considering the worst case, let's assume we only freeze one element. So now we have a derangement of size $N-1$, and we expect to perform $N - 2$ swaps before freezing. Thus we have the sum $N - 1 + N - 2 + ... = \binom{N}{2}$

Thus the expected running time of the random swapping algorithm is $\Theta(N^2)$. Although we have these worst and best case lower bounds $\frac{N^2}{2}$ and $\frac{N^2}{8}$, through simulations we observed that it takes almost exactly $\frac{N^2}{4}$ swaps. In the next sections, we derive a recurrence to get a more exact expected running time.

3.2 Recurrence for a more exact running time $N^2/4$

Observe that performing a swap freezes 0, 1, or 2 elements. Thus we can write a recurrence for the expected number of swaps to sort an derangement of size

N.

$$
\begin{aligned}
T(N) &= p_0 T(N) + p_1 T(N-1) + p_2 T(N-2) \\
T(N) - p_0 T(N) &= p_1 T(N-1) + p_2 T(N-2) \\
T(N)(1-p_0) &= p_1 T(N-1) + p_2 T(N-2) \\
T(N) &= \frac{p_1 T(N-1) + p_2 T(N-2)}{1-p_0} \\
T(N) &= \frac{p_1 T(N-1) + p_2 T(N-2)}{p1 + p2}
\end{aligned} \tag{2}
$$

Where p_1 and p_2 are the probabilities of a 1-swap and a 2-swap, respectively. So how can we find p_1 and p_2? Let's start by looking for p_2. Let's say we have a derangement A of size N and we pick i and j. If (i, j) is a 2-swap, that is, $A[i] = j$ and $A[j] = i$. We know the position of two elements and the rest of the $N - 2$ elements are deranged. So $p_2 = \frac{!(N-2)}{!N}$. Since the number of derangements of size N is approximately $\frac{N!}{e}$, then $p_2 \approx \frac{1}{N(N-1)}$. Now we try to find p_1. Let g be the number of good swaps, g_1 be the number of 1-swaps and g_2 be the number of 2-swaps. We have

$$
\begin{aligned}
E[g] &= E[g_1] + E[g_2] \\
N - \frac{1}{2} &= E[g_1] + \frac{1}{2} \\
E[g_1] &= N - 1
\end{aligned} \tag{3}
$$

We have

$$
\begin{aligned}
p_1 * \binom{N}{2} &= E[g_1] \\
p_1 &= E[g_1] * \frac{2}{N(N-1)} \\
p_1 &= (N-1) * \frac{2}{N(N-1)} \\
p_1 &= \frac{2}{(N-1)} - \frac{2}{N(N-1)} \\
p_1 &= \frac{2N}{N(N-1)} - \frac{2}{N(N-1)} \\
p_1 &= \frac{2}{N}
\end{aligned} \tag{4}
$$

Plugging in these values for p_1 and p_2 in the recurrence and running it shows that $T(N) \approx \frac{N^2}{4}$.

Now observe that to calculate p_1 and p_2 we assumed we started with any random derangement with equal probability. If we, for example, perform a

$2 - swap$, it is expected that any derangement of size $N - 2$ is equally likely to be a result of this swap.

3.3 swaps within derangements of size n lead to stable distribution - degree argument

here we attempt to explain why although derangements don't behave uniformly like permutations, maybe we can assume some sort of uniformity. show how we find that the degree, also known as the number of 0 swaps in a derangement, is $\binom{n}{2} - n + l_2$, we see the degree is dominated by the $\binom{n}{2}$ term

3.4 other mentions?

the degree from n - 1 derangements to n derangements is uniform: n * (n - 1)
the degree from n - 2 derangements to n derangements is uniform: n * (n - 1) / 2

4 Adversary

4.1 describing the adversary

Consider an adversary that tries to slow a blind sorting algorithm by delaying, as much as possible, any element from freezing. That is, if an algorithm performs a swap (i, j), the adversary will try to claim that $A[i] \neq j$ and $A[j] \neq i$. Of course, with enough swaps the adversary can no longer claim that an element doesn't belong in a certain index. For example, if an element has visited $N - 1$ indices and still hasn't been frozen, then we know it must belong in that 1 other index.

With that, let's define a bipartite graph that the adversary will maintain $G(V, E)$ where the vertex set V is split into two sets $L = l_1, l_2, ..., l_n$ and $I = i_1, i_2, ..., i_n$. The set L represents the elements of the array and the set I represents the indices of the array. We say an edge exists from vertex l to vertex i if the element l can possibly be in index i.

When an algorithm performs a swap (i, j) the adversary will say element $l = A[i]$ can not be in index j. Thus, the edge going from l to j will be removed. The analagous edge for element $A[j]$ will also be removed.

In this bipartite graph, there must be a (maximum) matching at all times between the set of elements and the set of indices. This matching represents a permutation of the array that is consistent with the swaps performed so far. When a swap is requested, before removing the edge, we must determine if after removing the edge if there is still a possible matching. If there isn't, then the adversary can not claim that the element doesn't belong in this position. Therefore, it must report to the algorithm that the element is frozen. In the next section, we will see how to implement this idea.

4.2 Implementation

```
from collections import defaultdict

class Adversary:
    def __init__(self, n):
        # no derangements for n < 2
        assert not n < 2

        # build graph
        self.elements = ['element{}'.format(i) for i in range(n)]
        self.graph = defaultdict(set)

        for i, element in enumerate(self.elements):
            for j in range(n):
                if i != j:
                    self.graph[element].add(j)

                if (i + 1) % n == j:
                    self._reverse_edge(element, j)

    def _reverse_edge(self, u, v):
        self.graph[u].remove(v)
        self.graph[v].add(u)

    def _is_in_matching(self, element, idx):
        return self.graph[idx] == {element}

    def _exist_path(self, u, v):
        seen = set()
        path = []

        def dfs(node):
            if node == v:
                return True

            if node not in seen:
                seen.add(node)
                for adj in self.graph[node]:
                    path.append((node, adj))
                    found = dfs(adj)
                    if found:
                        return True
                    path.pop()

            return False
```

```
        return dfs(u), path

    def _attempt_remove(self, idx, element):
        # if edge is not in matching, simply remove
        if not self._is_in_matching(element, idx):
            if idx in self.graph[element]:
                self.graph[element].remove(idx)
            return True

        # try to remove, and see if there is an alternating path
        self.graph[idx].remove(element)
        found, path = self._exist_path(element, idx)
        if found:
            for u, v in path:
                self._reverse_edge(u, v)

            return True

        # no alternating path found, edge cannot be removed.
        # add edge back into graph
        self.graph[idx].add(element)
        return False

    def swap(self, i, j):
        # perform swap
        self.elements[i], self.elements[j] = self.elements[j], self.elements[i]

        # initialize return value
        frozen = []

        # can we say elements[i] cannot be in position i
        removed = self._attempt_remove(i, self.elements[i])
        if not removed:
            # if can't remove, add it to frozen
            frozen.append(i)

        # repeat for j
        removed = self._attempt_remove(j, self.elements[j])
        if not removed:
            frozen.append(j)

        return frozen

    def original_array(self):
        n = len(self.elements)
```

```
        arr = [None] * n
        for i in range(n):
            element = next(iter(self.graph[i]))
            element_number = int(element.strip('element'))
            arr[element_number] = i

        return arr
```

Implementation of the deterministic algorithm that plays against the adversary

```
from adversary import Adversary

def deterministic(n):
    A = Adversary(n)
    frozen = set()
    swaps = 0

    for i in range(n):
        j = i + 1
        while i not in frozen:
            if j not in frozen:
                frozen.update(A.swap(i, j))
                swaps += 1
            j += 1

    print(A.original_array())
    return swaps
```

When the determinstic algorithm plays against the adversary for $N = 10$, the result of

```
print(A.original_array())
```

is

```
[6, 7, 8, 9, 10, 5, 4, 3, 2, 1]
```

which is consistent with the worst case input we attempted to build earlier (along with the number of swaps).

4.3 n^2 lower bound

All elements are frozen when the matching in our graph is unique. That is, if we remove an edge in the matching from the graph, then no other matching can be found. Let's consider two edges in the final matching (u, v) and (k, l). We know that at least the edge (u, l) or the edge (v, k) must have been removed from the graph, otherwise the matching is not unique. That is, for any pair of two edges

(we have $\binom{N}{2} - N$ pairs, since initially we are missing N edges) , we know at least 1 edge has been removed. Since a swap removes at most 2 edges, then at least $\frac{\binom{N}{2}-N}{2}$ swaps have been performed, thus establishing a lower bound of $\Omega(N^2)$ swaps for any Blind sorting algorithm.