# STRATEGIC ONLINE PLACEMENT: AN ALGORITHM FOR REAL-TIME, DISTRIBUTION-INFORMED POSITIONING

PABLO RUIZ CUEVAS

Abstract. In numerous systems, from manufacturing to aviation and from data management to emergency response, the challenge arises to position or sort incoming items in real-time based on their inherent characteristics and without the luxury of rearrangement. This paper introduces the Strategic Online Placement Algorithm (SOPA), a novel approach that leverages prior distribution knowledge to make optimal placement decisions as items arrive. Unlike traditional sorting algorithms, SOPA is designed for scenarios where items, once placed, cannot be moved. We demonstrate the broad applicability of SOPA across various domains and show that, with accurate distribution information, our method is the optimal placement strategy for the given information.

## 1. Introduction

In numerous systems spanning manufacturing, aviation, data management, and emergency response, there emerges a unique challenge: how to optimally position or sort incoming items in real-time, based on their inherent characteristics, and without the ability to reposition them later. Traditional sorting algorithms, which operate by comparing and potentially swapping elements, may not be suitable or efficient for these scenarios. This paper introduces the Strategic Online Placement Algorithm (SOPA), a novel approach that leverages prior distribution knowledge to make optimal placement decisions as items arrive, ensuring efficient system operations and improved outcomes.

### 1.1. **Related Work:.**

Online sorting and placement algorithms have a rich history, with many algorithms designed for specific scenarios such as stream processing, memory management, and real-time systems. Traditional online algorithms, like the ones proposed by Kushwaha et al. (2021), focus on adaptively sorting data but often rely on the possibility of rearranging items. Other works, such as those by [Author of the Mapping Sorting Algorithm paper et al., Year], have touched upon non-adaptive sorting but without the incorporation of prior distribution knowledge. A notable advancement in this domain is the work of Chaikhan et al. (2022) who proposed a fast continuous streaming sort algorithm for big streaming data environments under fixed-size single storage. This algorithm particularly addresses the constraint of storage overflow in real-time systems, which resonates with the problem scenario of fixed placement discussed in this paper. However, the existing literature including the work of Chaikhan et al. (2022) does not delve into leveraging prior distribution knowledge for optimal item placement, which is the focal point of this paper. Clearly, a gap exists for an approach that combines real-time placement, non-adaptiveness, and distribution-aware decision-making.

### 1.2. **Problem Definition:.**

Given a stream of N items arriving sequentially, the objective is to place each item in a fixed position within an array of length N based on its characteristics. Two primary constraints define this problem:

- Online Processing: Items are processed in the order they arrive, without knowledge of future items.
- Non-adaptive Placement: Once an item is placed in a position, it cannot be moved.

The placement decisions should leverage known or estimated distributions of the items' characteristics to achieve optimal or near-optimal sorting. The "optimality" in this context can be defined based on specific goals, such as minimizing disruptions in a sequence, ensuring balanced weight distribution, or any other domain-specific objective.

We will solve the problem for the case where:

- Continuous case where $N \sim U(0, 1)$
- Discrete case without replacment where $N \sim U\{1, 2, ..., M\}$

This cases can be extended without loss of generalitly to any other distribution, as we will proof.

In this paper we will show the best posible algorithm/strategy to maximize the probability of sorting the array, as well as the probability of succesfully sorting the array using the optimal algorithm. We will refer to this probability as the **Optimal Success Rate** (OSR)

### 1.3. **Continuous Case.**

**Theorem 1.3.1.** *The Probability of sorting a list of $n$ numbers uniformly distributed between 0 and 1 using optimal strategy for maximizing the probability of getting the array sorted is:*

$$P_n = \int_0^1 \sum_{k=1}^{n} P_{nk} dn_1 \tag{1}$$

$$P_{nk}(n_1) = \begin{cases} 0 \; if \; \left(P_{nk}(n_1) < P_{n(k_2 \neq k)}(n_1)\right) \wedge k \notin \{0, n\} \\ Bin(n-1, k-1, n_1) P_{n-1-k} P_k \; else \end{cases} \tag{2}$$

*Where $P_{nk}(n_1)$ is the OSR of an $n$ elements array array if we starting with $n_1$ at the position $k$. And Bin is the binomial distribution, with a success probability $P[n_2 < n_1] = n_1 = p$ as $n \in (0, 1]$*

$$Bin(n, k, p) = \binom{n}{k} p^{n-k} p^k \tag{3}$$

*Proof.* Having the Optimal Sucess Rate (OSR) for $n < m$ slots, the OSR of $m$ slots is the expected value of optimally placing $n_1$ for each possible $n_1$.

Placing the first number $n_1$ at the $k$ slot, divides the problem in two subdomains of itself with domains $[0, k-1]$ and $[k+1, 1]$. Where the probability of having all the subsequent numbers correctly fitting in the new domains is given by a binomial distribution with $n-1$ trials, $k-1$ needed sucesses and a probability of succes $p = n_1$ . □

The solution of the problem relies in the naive solution of $P_1 = 1$ for calculating the OSR for $n > 1$ with the recursive formula which implements a "divide and conquer" strategy.

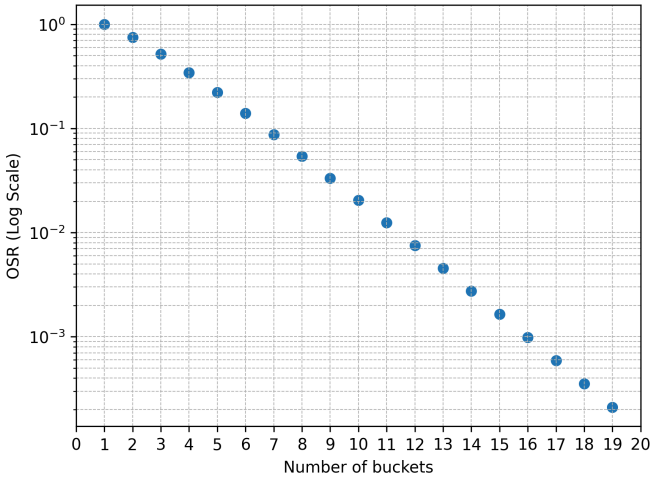


FIGURE 1. Optimal Success Rate for $n$ slots, as seen in the figure, the probabilities of sorting it at once decrease in a exponential fashion.

The OSR can always be found as an analytical solution, as all of the terms can be expressed in terms of polinomios and operations to polinomios.

In practice, for computing the OSR we can alternatively write the function by spliting the integral by the optimal ranges and defining $p_{nk}(n_1)$ as the probability of sorting the array given that we place the first element at the position $k$, so the solution will be given by the next formula:

$$P_n = \sum_{k=1}^{n} \int_{O_{nk}}^{O_{nk+1}} p_{nk} dn_1 \tag{4}$$

With:

$$p_{nk}(n_1) = \text{Bin}(n-1, k-1, n_1) P_{n-1-k} P_k \tag{5}$$

$$O_{nk} = \begin{cases} 0 \text{ if } k = 1 \\ \text{where } p_{nk}(n_1) = p_{n(k+1)}(n_1) \\ 1 \text{ if } k = n \end{cases} \tag{6}$$

We can see the representation of this solution in the Figure 2, for the case $n = 7$.

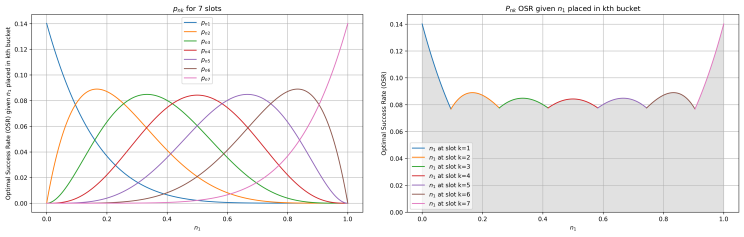


FIGURE 2. Optimal Success Rate for $n$ slots, as seen in the figure, the probabilities of sorting it at once decrease in a exponential fashion.

### 1.4. **The Sorting algorithm.**

Once we know the OSR discussed in the previous section, we can create a probabilistic algorithm for sorting an array in a online, non-adaptive way.

Given $n_1$ we only need to place it where $p_{nk}(n_1)$ is maximun and for the next $n_m$ we have two posibilities:

- $n_m$: can be placed in the new array, without breaking the order, then we normalize the n_m using the extremes where it fits: $t_m = \frac{n_m - \min}{\max - \min}$
- $n_m$: can not be placed in the new array. In this case the algorithm has failed in sorting but we can still get the best possible sort by ignoring the placed values and use the same algorithm with the empty slots.

Given the number of slots $n$ and the first number to place $n_1$ we place it by the index given by the optimal threshold $k$. Given $n_2$ we run the same algorithm but in the range $(k, n] \forall n_2 > n_1$ or $[0, k) \forall n_2 < n_1$ normalizing $n_2$ as $t_2 = \frac{n_2 - \min}{\max - \min}$, where max and min are the biggest and smallest number we can fit in the subproblem.

For the next turns, there is the posibility that the index suggested by the threshold is already taken, in that case we will move the number in the direction that

preserves the order as far as we need, but when there is no possible fit we have different posibilities that will further define the algorithm:

- Ignore all items already in place and run the algorithm for the empty slots.
- Try to place it in the neighbourhoud of the index that it corresponds, using some heuristics:
    - For instance we could go to the right or left till we find a place and if no place is found then switch the direction.
    - We could alternate left and right directions in an increasing pattern 1,-2,+3,-4 etc.
    - We could check wich option of the left or right placing let's the whole array in a more skewed postion, measuring center of mass for instance.

The problem of the heuristical approaches is that they don't consider the impact in the later game.

For $n_2$ if $n_2$ is bigger than $n_1$ we run the same algorithm but in the intre

**Algorithm 1:** Algorithm 1: Pseudo Sort

```
Create empty slots array
get n_1
Place it at k = I(n_1), slots[k]=n_1
get n_2 and k_2 = I(n_2)
PROCEDURE
IF slots[k_2] is empty then
    slots[k_2] = n_2
ELIF n_2 < slots[k_2]

Blinker aus
```

1.5. **prove of algorithm bestnes.**

What we in principle know about our algorithm is that it has the OSR, so the probabilities of sorting the array are maximized.

As the algorithm is recursive, this property is also satisfied at every subset of the array, and in case of failure to sort, as we choose to ignore placed items and apply the same algorithm, this property still applies.

But other properties remain still unclear in principle, specially for the case the algorithm it fails to do a perfect sort.

1. Are there algorithms that on average have higher Correctly Placed Items? (ACPI)
2. Are there algorithms that on average take more Turns till Failure (ATF)
3. Are there algorithms that results on average in a higher center of mass (ACM)

4. Are there algorithms that results on average in a smaller distance to correct position (AD)
5. Are there algorithms that results on average in a smaller average max distance to correct position (AMD)?
6. Are there algorithms that on failure case sort better, in terms of center of mass etc?

ACPI, We know that for $n = 2$ the answer is trivially no, When we are in the position of having failed and rerun the algorithm in the subset we do

For the rest of the points let us explore the case of $n = 2$ in that case is trivial to prove that the algorithm porpoused maximices CM and ICP at the time it minimices AD and AMD as it maximizes the probability of the two numbers being sorted.

For the case $n = 3$ We need to think again in terms of the possibilities:

we can prove by contradiction most of the properties, imagine that there is an algorithm that in case of failure does have ha better ACPI, ATF, ACM or AD, that would mean that that algorithm would also be be

## References

Department of my room, My self, Munich, 80636
*Email address:* pablo.r.c@live.com
*URL:* www.math.sc.edu/~howard