Cryptography Meets Algorithms (15893) Lecture Notes

Lecture 9: Oblivious Sorting

Scribe: Ying Feng

February 28, 2024

We will cover oblivious sorting in today's lecture, which is a powerful building block in the design of many oblivious algorithms. In an earlier lecture, we learned about how to construct Oblivious RAM (ORAM). ORAM lets you compile any non-oblivious algorithm into an oblivious counterpart. However, for a specific computation task, using ORAM to generically compile an existing non-oblivious algorithm often is not the fastest approach. Today we will see one such example, namely, oblivious sorting. In future lectures, we will see more examples, such as oblivious data structures, where customized oblivious algorithms asymptotically outperform generic ORAM-compilation of an existing non-oblivious algorithm.

1 Oblivious Sorting

Definition 1. An algorithm $\textsc{Alg}$ is **oblivious** if for all pairs of inputs $\mathcal{I}, \mathcal{I}'$ of equal length, we have:

$$\textsc{AccessPattern}(\textsc{Alg}, \mathcal{I}) \approx \textsc{AccessPattern}(\textsc{Alg}, \mathcal{I}'),$$

where $\textsc{AccessPattern}(\textsc{Alg}, \mathcal{I})$ denotes the observed access pattern when executing $\textsc{Alg}$ on input $\mathcal{I}$, and $\approx$ denotes two distributions are indistinguishable (either statistically or computationally, depending on the specific setting.)

Non-Example 2. QuickSort is **not** oblivious because it reads and writes array elements depending on the result of comparing with a pivot. These instructions leak information about the content of the input array.

Consider the following two arrays:

$$Arr_1 := \begin{array}{|c|c|c|c|c|c|c|}\hline 7 & 6 & 5 & \mathbf{4} & 3 & 2 & 1 \\ \hline\end{array}$$

$$Arr_2 := \begin{array}{|c|c|c|c|c|c|c|}\hline 1 & 2 & 3 & \mathbf{4} & 5 & 6 & 7 \\ \hline\end{array}$$

If in both Arr_1 and Arr_2, **4** is picked to be the pivot, then QuickSort would move $Arr_1[0:2]$ to the right of **4** while keeping $Arr_2[0:2]$ at the left of **4**. Therefore, one can distinguish the two inputs Arr_1 and Arr_2 by observing the different access patterns.

We can naïvely compile QuickSort to be oblivious using $\textsc{PathOram}$. However, this incurs a log-squared (multiplicative) overhead. Since the runtime of QuickSort is $\mathcal{O}(n \log n)$ where n is the input length, the resulting oblivious version of QuickSort runs in $\mathcal{O}(n \log^3 n)$.

In this lecture, we will see two direct constructions of oblivious sorting that achieve faster runtime than compiling QuickSort with ORAM. Our ultimate goal is to match the runtime of non-oblivious QuickSort, i.e. $\mathcal{O}(n \log n)$.

Remark 1. The first construction that achieves the desired $\mathcal{O}(n \log n)$ performance is the AKS sorting network [AKS83]. However, the AKS network is not used in practice due to the large constant factor hidden by the Big-O notation.

2 Bitonic Sort

We first describe the famous bitonic sort algorithm by Batcher [Bat68].

Definition 3. A sequence $A = [A_0, A_1, \cdots, A_n]$ is **bitonic** if and only if:

1. There exists some $i \in [n]$ such that $A_j \leq A_{j+1}$ for all $j < i$, and $A_j \geq A_{j+1}$ for all $j \geq i$, i.e.,

$$A_0 \leq A_1 \leq \cdots \leq A_i \geq A_{i+1} \geq \cdots \geq A_n,$$

   or

2. There exists a cyclic shift π on A such that $\pi(A)$ satisfies the above requirement 1.

We say a bitonic sequence is in the *canonical form* if it satisfies requirement 1.

We specify the BitonicSort algorithm as multiple components. We start with introducing the following BitonicSplit operation:

Algorithm 1: BitonicSplit

```
Input: $A$, a bitonic sequence of length $n$.
Output: $(L_A, R_A)$, a pair of two sequences each of length $n/2$.
$L_A \leftarrow [\min(A_0, A_{\frac{n}{2}}), \min(A_1, A_{\frac{n}{2}+1}), \cdots, \min(A_{\frac{n}{2}-1}, A_{n-1})]$;
$R_A \leftarrow [\max(A_0, A_{\frac{n}{2}}), \max(A_1, A_{\frac{n}{2}+1}), \cdots, \max(A_{\frac{n}{2}-1}, A_{n-1})]$;
return $(L_A, R_A)$;
```

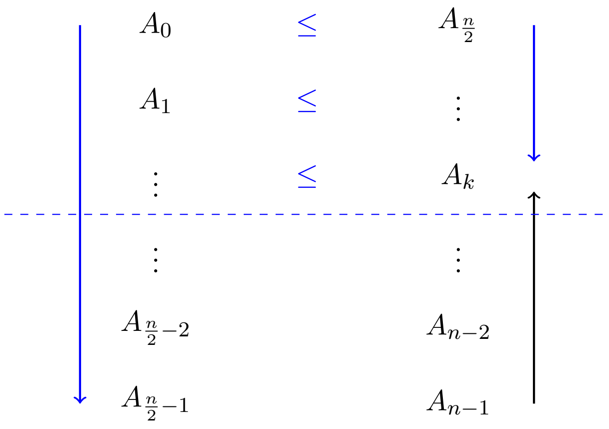
Claim 4. L_A and R_A are bitonic, and $\max_i(L_A[i]) \leq \min_j(R_A[j])$.

Remark 2. We will prove Claim 4 for canonical-form bitonic sequences. However, in fact the claim also holds for any bitonic sequence that are not necessarily in the canonical form. From an asymptotical point of view, assuming canonical form is without loss of generality because any bitonic sequence can be converted into its canonical form in linear time by applying the corresponding cyclic shift π. So the asymptotical performance of the sorting algorithm is unaffected.

Proof of Claim 4. As mentioned above, we assume that A is in canonical form. We denote the index of the "turning point" as k, such that

$$A_0 \leq A_1 \leq \cdots \leq A_k \geq A_{k+1} \geq \cdots \geq A_n.$$

Assuming without loss of generality that $k \geq n/2$ (the case of $k < n/2$ is symmetric), we can visualize the sequence as the following two columns:

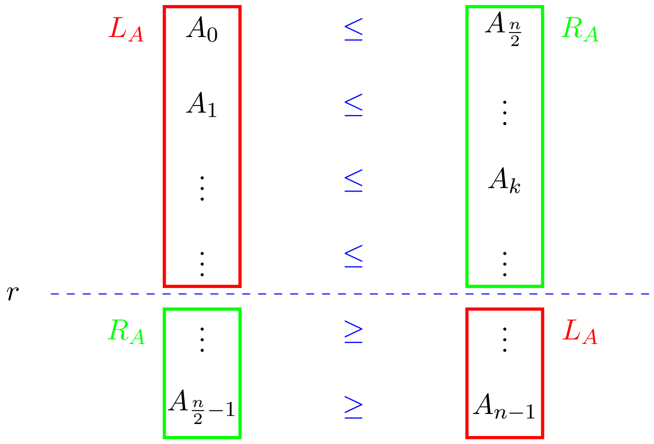


where the blue and black arrows point to the larger values. Note that in the top $k - n/2 + 1$ rows (until the dashed line), the value on the right column must be greater than or equal to that on the left column.

Moreover, starting from the $(k - n/2 + 2)^{\text{th}}$ row (i.e., below the dashed line), if we gradually move down the dashed line, we will find a unique row r (or reach the end) such that:

- Above this row, the value on the right column is greater than or equal to that on the left column, and
- Below this row, the value on the right column is less than or equal to that on the left column.

This is visualized as follows:

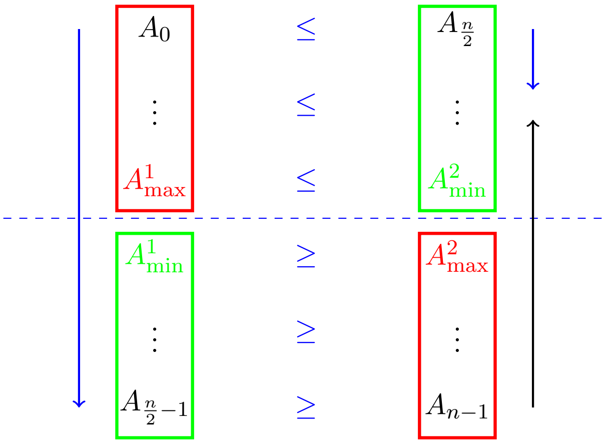


After BITONICSPLIT, L_A is composed of the elements in the red brackets, and R_A is composed of the elements in the green brackets. Therefore, both L_A and R_A are in fact **subsequences** of the original bitonic sequence A, which must be bitonic.

Lastly, we argue that $\max_i(L_A[i]) \leq \min_j(R_A[j])$. Observe that

$$\max_i(L_A[i]) = \max(A^1_{\max}, A^2_{\max}) \text{ and } \min_j(R_A[j]) = \max(A^1_{\min}, A^2_{\min})$$

in the following figure.



We have that:

$$A^1_{\max} \le A^1_{\min} \text{ and } A^2_{\max} \le A^2_{\min} \qquad \text{as shown by the directions of arrows}$$
$$A^1_{\max} \le A^2_{\min} \text{ and } A^2_{\max} \le A^1_{\min} \qquad \text{as shown by the signs between columns}$$

Therefore $\max_i(L_A[i]) = \max(A^1_{\max}, A^2_{\max}) \le \min_j(R_A[j]) = \max(A^1_{\min}, A^2_{\min})$.

□

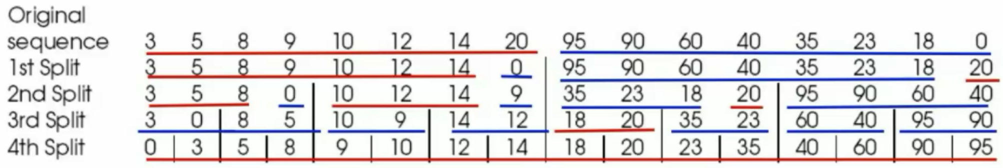
With BITONICSPLIT as a subroutine, we can now specify the following recursive merging step and the general BITONICSORT algorithm. We use $||$ to denote the concatenation of two sequences and use $\neg$ to denote flipping the direction, i.e., $\neg dec = inc$ and $\neg inc = dec$.

Algorithm 2: BITONICMERGE

```
Input: $(A, \text{direction})$
$A$ is a bitonic sequence of length $n$ and direction $\in \{inc, dec\}$.
Output: $result$, a sorted sequence.
if $|A| = 1$ then
    return $A$;
$(L_A, R_A) \leftarrow$ BITONICSPLIT$(A)$;
if direction $= dec$ then
    Swap $(L_A, R_A)$;
return BITONICMERGE$(L_A, \text{direction})$ || BITONICMERGE$(R_A, \text{direction})$;
```

In summary, BITONICMERGE sorts a sequence under the assumption that it is bitonic. Claim 4 guarantees that the assumption is maintained during the recursions. Finally, in order to sort a general array, we need to firstly preprocess it to be bitonic. This is again done by recursion.

A concrete example of the recursion is given as follows:



Original sequence	3	5	8	9	10	12	14	20	95	90	60	40	35	23	18	0
1st Split	3	5	8	9	10	12	14	0	95	90	60	40	35	23	18	20
2nd Split	3	5	8	0	10	12	14	9	35	23	18	20	95	90	60	40
3rd Split	3	0	8	5	10	9	14	12	18	20	35	23	60	40	95	90
4th Split	0	3	5	8	9	10	12	14	18	20	23	35	40	60	90	95

Algorithm 3: BITONICSORT

```
Input: $(A, \text{direction})$
$A$ is a sequence of length $n$ and direction $\in \{inc, dec\}$.
Output: $result$, a sorted sequence.
if $|A| = 1$ then
    return $A$;
$S_1 \leftarrow$ BITONICSORT($A[0, n/2 - 1]$, direction);
$S_2 \leftarrow$ BITONICSORT($A[n/2, n - 1]$, $\neg$direction);
return BITONICMERGE($S_1 \,||\, S_2$, direction);
```

Analysis. BITONICSORT is oblivious because the behavior of the algorithm does not depend on the content of the input, i.e., the sequence of comparisons is deterministic regardless of the comparison results. At each layer of the recursion, the total number of comparisons made by all calls in this layer is upper bounded by $n/2$. The depth of the recursion is given by

$$D(n) = D(n/2) + \log n$$

which solves to $\log n(\log n + 1)/2$. Therefore the total number of comparisons is upper bounded by $n/2 \cdot \log n(\log n + 1)/2 = \mathcal{O}(n \log^2 n)$, where the Big-O notation hides a relatively small constant.

3 Bucket Oblivious Sort

We start by introducing the Bucket Oblivious Sorting algorithm [ACN$^+$] that has an $\mathcal{O}(n \log n (\log \log n)^2)$ complexity. Afterward, we will discuss an improvement to speed it up to $\mathcal{O}(n \log n)$. In general, the framework of BucketOSort is as follows:

Given a sequence A:

1. Apply an **oblivious random permutation** (ORP) on A to get $A' \leftarrow \text{ORP}(A)$.
2. Sort A' using any (potentially non-oblivious) comparison-based sorting algorithm, e.g. QuickSort.

As the name suggested, the specification of ORP is that given a sequence A, outputs a randomly permuted version of A. The obliviousness guarantee is that the access pattern should reveal nothing about the applied random permutation. Our plan is to construct ORP as a randomized algorithm with a *deterministic access pattern.*

Non-Example 5. The Fisher–Yates shuffle is not an oblivious random permutation. It works as follows: Given $A := [A_0, \cdots, A_{n-1}]$,

- for $i \in [0, n-1)$:
  - $j \overset{\$}{\leftarrow} [i, n)$;
  - Swap(A_i, A_j);

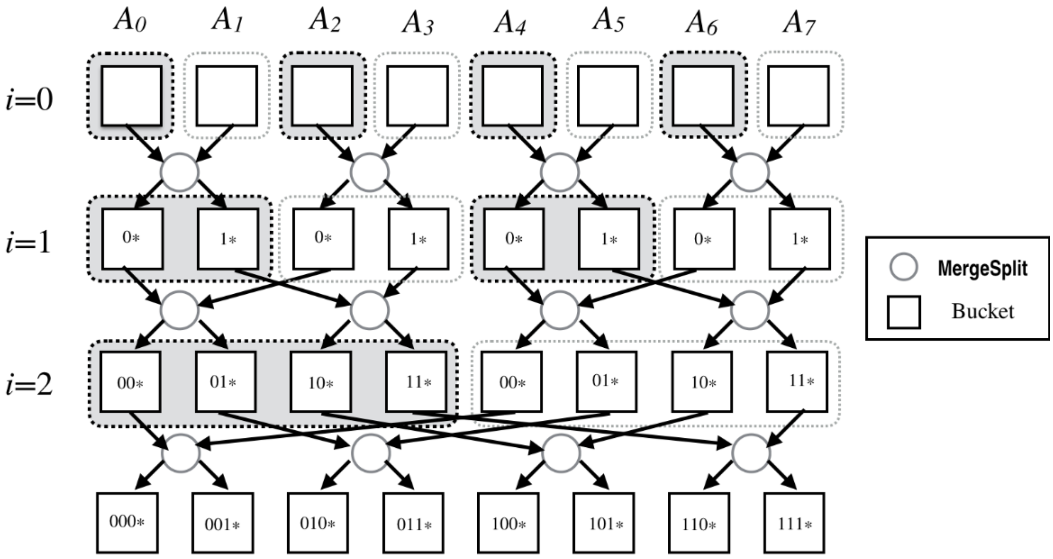
Apparently, one can exactly learn the random permutation applied by observing the Swap instructions.

Remark 3. It is crucial that in the second step of the BucketOSort framework, the sorting algorithm needs to be comparison-based. For example, we cannot use counting-based sort (where we just throw elements into bins based on their keys), because even after the sequence is randomly permuted, applying counting sort would reveal the frequency of elements in each bin, which is exactly what we want to hide. In contrast, a comparison-based sorting will only reveal the relative order between elements, which is fine after we randomly permuted the positions of elements.

3.1 Oblivious Random Permutation

For the purpose of construction ORP, we are going to assign each element to a random bucket and then route the elements through a butterfly network to their assigned random buckets. A visualization of the network is given on the next page. The behavior of the algorithm is as follows:

- (Set-up): Let Z be a size of buckets (which we will choose later as a function of n). Let $B := 2n/Z$ be the number of buckets. Each bucket is going to contain $Z/2$ *real* elements from the input sequence and $Z/2$ dummy elements.
- Each element is randomly assigned to one of the B buckets indexed by a key of $\log B$ bits.
- The elements are routed to their respective destinations through a butterfly network, which contains $\log B$ layers that are connected through MERGESPLIT operators.
- The MERGESPLIT operator takes two buckets from the i^{th} level as inputs and reorganize their elements into two buckets for the $(i+1)^{\text{th}}$ layer, according to the $(i+1)^{\text{th}}$ most significant bit of the elements' keys. i.e., At the i^{th} level, every $2i$ consecutive buckets are *semi-sorted* by the most significant i bits of the keys.
- Finally, to obtain a random permutation from our random bucket assignments, we obliviously and randomly permute the real elements within each bucket in the final layer. Further, we remove the dummy elements from each bucket in the finaly layer. The bucket-wise oblivious random permutation can be accomplished by assigning a random $\log n$-bit label to each real element in the bucket, and and then calling bitonic sort to sort the real elements by their labels, while all dummies at the end of the bucket (if two elements are found to have duplicate labels, we can repeat the process). Removing the dummies can be by making a pass over the final-layer buckets, writing down only the real elements. During this process, the number of dummies in each final-layer bucket is revealed — but this information is safe to reveal since it is simulatable without knowledge of the contents of the input array to be sorted.



Efficiency for clients with $2Z$ storage. If the client has a storage size no less than $2Z$, then they can perform MERGESPLIT and random permute the buckets at the final layer locally. In each of the $\log B = \log \frac{2n}{Z} < \log n$ layers, the $2n$-sized sequence (containing n real elements and n dummy elements) is read and written once. So oblivious bucket assignment and bucket ORP has $4n \log n$ cost (measured in terms of reads and writes to memory).

Efficiency for clients with constant storage. With constant-storage clients, we need to implement oblivious MERGESPLIT using BITONICSORT. Specifically, we can sort the two input buckets such that all real elements that want to go left are in the front, all real elements that want to go right are at the end, and some dummy elements can be in the middle. To bound the cost of all MERGESPLIT operations: observe that at each of the $\log B$ layers, we invoke $B/2$ instances of bitonic sort on $2Z$ elements, and thus the total cost is roughly

$$\log B \cdot B/2 \cdot 2Z \log^2(2Z) \approx 2n \log n \log^2 Z.$$

In particular, if we set $Z = \log^2 n$ which is sufficient for getting negligibly small failure probability (see Lemma 6), we have that the the above expression becomes $O(n \log n (\log\log n)^2)$.

The other costs, e.g., cost of preparing the input layer, the cost of bucket-wise permutation in the final layer, and the cost of dummy removal in the final layer, are all asymptotically dominated by the cost of the MERGESPLIT operations.

Success probability. The above algorithm is guaranteed to output a correct, randomly permuted version of the input sequence if none of the buckets ever overflows (i.e., receives more than Z real elements) during the routing.

Lemma 6. Overflow happens with at most $\epsilon(n, Z) = 2n/Z \cdot \log(2n/Z) \cdot e^{-Z/6}$ probability.

Proof. Consider a bucket B_i at level i which has an associated i-bit long prefix b of keys. B_i receives real elements from 2^i initial buckets in the first layer, each containing $Z/2$ real elements. By the construction, any of these elements reaches B_i only if the most significant i bits of its key match b, which happens with exactly 2^{-i} probability.

A Chernoff bound shows that B_i overflows with less than $e^{-Z/6}$ probability. Hence, taking a union bound over all levels and all buckets, we get that overflow happens with less than $\epsilon(n, Z) := B \cdot \log B \cdot e^{-Z/6}$ probability. □

Lemma 6 shows that it suffices to set Z to be any superlogarithmic function for the failure probability to be negligibly small.

4 Subsequent Improvements of Bucket Oblivious Sort

We can get rid of the extra $(\log\log n)^2$ factors using the following approaches. First, the work of Ramachadran and Shi [RS21] showed that by using a $\log n$-fanout butterfly network and using AKS to realize the $\log n$-way MERGESPLIT operator, they can achieve $O(n \log n)$ cost. Second, Asharov et al. [ACN+] showed that if we use linear-time oblivious tight compaction [AKL+20] to realize the MERGESPLIT operator, we can also get $O(n \log n)$ cost. However, neither of these approaches result in practical schemes.

The recent work of Gu et al. [GWC+23] showed a practical instantiation of Ramachandran and Shi's approach of using a $\log n$-fanout butterfly network [RS21]. Their approach is practical and achieves $O(n \log n)$ cost — specifically, they devised a new approach to efficiently realize a multi-way MERGESPLIT.

Finally, optimal oblivious heap [Shi20, JLS21] which we will cover in a future lecture also gives rise to a concretely efficient $O(n \log n)$ oblivious sorting algorithm. However, the resulting sorting algorithm may be less amenable to parallelization than the ones presented today.

References

[ACN+] Gilad Asharov, T-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. *Bucket Oblivious Sort: An Extremely Simple Oblivious Sort*, pages 8–14.

[AKL+20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious RAM. In *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 403–432. Springer, 2020.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, page 1–9, New York, NY, USA, 1983. Association for Computing Machinery.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery.

[GWC+23] Tianyao Gu, Yilei Wang, Bingnan Chen, Afonso Tinoco, Elaine Shi, and Ke Yi. Efficient oblivious sorting and shuffling for hardware enclaves. Cryptology ePrint Archive, Paper 2023/1258, 2023. `https://eprint.iacr.org/2023/1258`.

[JLS21] Zahra Jafargholi, Kasper Green Larsen, and Mark Simkin. Optimal oblivious priority queues. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2366–2383. SIAM, 2021.

[RS21] Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. In *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, pages 373–384. ACM, 2021.

[Shi20] Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 842–858. IEEE, 2020.