POLITECNICO
MILANO 1863

Constant Time, Parallel Shuffling

095947 - CRYPTOGRAPHY AND ARCHITECTURES FOR COMPUTER SECURITY

Conti Alessandro

Table of Contents

Specification

Shuffling arrays in constant time is a common problem in modern cryptography. This project involves analysing the technique proposed by Daniel J. Bernstein in `https://cr.yp.to/2024/insertionseries-20240515.py`, and implementing it in C, possibly employing parallelization.

Why Does It Matter?

In many cryptographic applications, it is necessary to perform multiple insertions within an array or list

- construction of constant-weight words used in the McEliece cryptosystem;
- insertion of a blockchain transaction in the mempool.

However, naive implementations is very slowly and may expose data to side-channel attacks. If memory access depends on data, the adversary may infer sensitive information.

Inserting an Element into an Array

To insert an element at a specific position in an array of size n, it is necessary:

- increase the size of the array if all positions are full, $\mathcal{C}(n) = \mathcal{O}(n)$;
- move all elements to the right of the specified position by one position, $\mathcal{C}(n) = \mathcal{O}(n)$;
- insert the new element at the desired, free position, $\mathcal{C}(n) = \mathcal{O}(1)$.

This algorithm has the following computational cost

$$\mathcal{C}_{insert}(n) = \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$$

Inserting Multiple Elements into an Array

To insert m elements in specific positions in an array of size n, it is necessary to repeat for m times the insertion of a single element.

This algorithm has the following computational cost

$$\mathcal{C}_{multiple\ \ insertions}(n, m) = m \cdot \mathcal{C}_{insert}(n) = \mathcal{O}(m \cdot n)$$

which is highly inefficient if most of the entries in an array are multiple entries and not single entries.

Insertion Series Objective

We will now analyse an algorithm that attempts to optimise serial insertion within an array. Trying to go from having a quadratic complexity to a quasi-linear one.

We will analyse all the complexities of the auxiliary functions until we arrive at the actual complexity of the algorithm.

Insertion Series Approach

The algorithm shown below is based on the idea of the mergesort algorithm.

- Divide-et-Impera
  - recursively sorts the array of $\langle position, element\rangle$ to be inserted.
- Sorted Merge
  - performs a smart merge between the destination array and the insertion array.

PrefixSum Function


```python
def prefixsums(L):
    result = [0]
    for b in L:
        result += [result[-1] + b]
    return result
```

PREFIXSUM Function (2)

Algorithm prefixSum Function

Input:

array: the array whose cumulative prefixes must be calculated

Output:

result: the cumulative prefixes of the array

```
function PREFIXSUM(array)
    result ← [ ]
    sum ← 0
    for i = 0 to array.size do
        result[i] ← sum
        sum ← sum + array[i]
    end for
    return result
end function
```

PREFIXSUM Example

Example

Consider the following array:

$$array = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

The prefixSum array is:

$$\begin{aligned} result &= \begin{bmatrix} 0 & 0+1 & (0+1)+2 & ((0+1)+2)+3 & (((0+1)+2)+3)+4 \end{bmatrix} = \\ &= \begin{bmatrix} 0 & 1 & 3 & 6 & 10 \end{bmatrix} \end{aligned}$$

PREFIXSUM Cost

Let $n := array.size$.

$$
\begin{aligned}
\mathcal{C}(n) &= \mathcal{C}_{intlist\_init}(n+1) + \mathcal{C}_{intlist\_reserve}(n+1) + (n+1)\cdot\mathcal{C}_{intlist\_append}(n+1) = \\
&= 1 + (n+1) + (n+1)\cdot 1 = \\
&= 2\cdot n + 2 = \\
&= \mathcal{O}(n)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}(n) &= \mathcal{S}_{IntList}(n+1) + \mathcal{S}_{int} = \\
&= ((n+1)\cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + 4\,\mathrm{B} = \\
&= 4\,\mathrm{B}\cdot n + 24\,\mathrm{B} = \\
&= \mathcal{O}(n)
\end{aligned}
$$

MERGE Function


```python
def merge(L, R):
    return sorted(L + R)
```

MERGE Function (2)

Algorithm merge Function

Input:

*array*1: the first array that needs to be merged

*array*2: the second array that needs to be merged

Output:

result: the ordered union of the two input arrays

```
function MERGE(array1, array2)
    merged ← [ ]
    merged ← CONCATENATE(array1, array2)
    result ← SORT(merged)
    return result
end function
```

MERGE Cost

Let $n := array1.size$ and $m := array2.size$.

$$
\begin{aligned}
\mathcal{C}(n, m) &= \mathcal{C}_{malloc}(n+m) + \mathcal{C}_{memcpy}(n) + \mathcal{C}_{memcpy}(m) + \\
&\quad + \mathcal{C}_{sort}(n+m) = \\
&= 1 + n + m + \\
&\quad + \left(\frac{n+m}{2} \cdot \log(n+m) \cdot \frac{\log(n+m)+1}{2}\right) = \\
&= \mathcal{O}\left((n+m) \cdot (\log(n+m))^2\right)
\end{aligned}
$$

For the sorting algorithm, the **bitonic sorting** network algorithm was chosen, with adaptations to handle arrays whose sizes are not perfect squares.
This choice is due to the fact that bitonic sort is an algorithm that can be easily parallelised and maintains a fixed sequence of comparisons and exchanges for the same array length. This makes the algorithm resistant to side channel attacks, as the control flow is independent of the data.

MERGE Cost (2)

$$
\begin{aligned}
\mathcal{S}(n, m) &= \mathcal{S}_{size\_t} + \mathcal{S}_{*Quadruple}(n + m) = \\
&= 8\,\mathrm{B} + (n + m) \cdot 16\,\mathrm{B} = \\
&= 16\,\mathrm{B} \cdot (n + m) + 8\,\mathrm{B} = \\
&= \mathcal{O}(n + m)
\end{aligned}
$$

INSERTIONSERIES_SORT_MERGE Function


```python
def insertionseries_sort_merge(L, R):
    L = [(x, 1, 0, y) for j, (x, y) in enumerate(L)]
    R = [(x - j, 0, j, y) for j, (x, y) in enumerate(R)]
    M = merge(L, R)
    offsets = prefixsums(1 - fromL for _, fromL, _, _ in M)
    return [(x + offset, y) for (x, _, _, y), offset in zip(M, offsets)]
```

INSERTIONSERIES_SORT_MERGE Function (2)

Algorithm insertionseries_sort_merge Function

Input:

tuples1: the first array of tuples that needs to be merged
tuples2: the second array of tuples that needs to be merged

Output:

result: the ordered union of the two input array of tuples

```
function INSERTIONSERIESSORTMERGE(tuples1, tuples2)
    quadruples1 ← [ ]
    quadruples2 ← [ ]
    for i = 0 to tuples1.size do
        ⟨position, element⟩ ← tuples1[i]
        quadruples1[i] ← ⟨position, element, fromT1, 0⟩
    end for
    for j = 0 to tuples2.size do
        ⟨position, element⟩ ← tuples2[j]
        quadruples2[i] ← ⟨position − j, element, fromT2, j⟩
    end for
    quadruplesMerged ← MERGE(quadruples1, quadruples2)
    inverses ← [ ]
    for i = 0 to quadruplesMerged.size do
        ⟨_, _, fromTuple, _⟩ ← quadruplesMerged[i]
        inverses[i] ← 1 − fromTuple
    end for
    offsets ← PREFIXSUM(inverses)
    result ← [ ]
    for i = 0 to quadruplesMerged.size do
        ⟨key, value, _, _⟩ ← quadruplesMerged[i]
        result[i] ← ⟨key + offsets[i], value⟩
    end for
    return result
end function
```

INSERTIONSERIES_SORT_MERGE Cost

Let $n := tuples1.size$ and $m := tuples2.size$.

$$\begin{aligned}
\mathcal{C}(n,m) &= \mathcal{C}_{malloc}(n) + \mathcal{C}_{malloc}(m) + n + m + \mathcal{C}_{merge}(n,m) + \mathcal{C}_{intlist_init}(n+m) + \\
&\quad + \mathcal{C}_{intlist_reserve}(n+m) + (n+m)\cdot\mathcal{C}_{intlist_append}(n+m) + \\
&\quad + \mathcal{C}_{prefixSum}(n+m) + \mathcal{C}_{pairlist_init}(n+m) + \mathcal{C}_{pairlist_reserve}(n+m) + \\
&\quad + (n+m)\cdot\mathcal{C}_{pairlist_append}(n+m) + \mathcal{C}_{free}(n) + \mathcal{C}_{free}(m) + \mathcal{C}_{free}(n+m) + \\
&\quad + \mathcal{C}_{intlist_free}(n+m) + \mathcal{C}_{intlist_free}(n+m+1) = \\
&= 1 + 1 + n + m + \left(1 + n + m + \left(\frac{n+m}{2}\cdot\log(n+m)\cdot\frac{\log(n+m)+1}{2}\right)\right) + \\
&\quad + 1 + (n+m) + (n+m)\cdot 1 + (2\cdot(n+m)+2) + 1 + (n+m) + (n+m)\cdot 1 + \\
&\quad + 1 + 1 + 1 + 1 + 1 = \\
&= 8\cdot(n+m) + \left(\frac{n+m}{2}\cdot\log(n+m)\cdot\frac{\log(n+m)+1}{2}\right) + 12 = \\
&= \mathcal{O}\Big((n+m)\cdot(\log(n+m))^2\Big)
\end{aligned}$$

INSERTIONSERIES_SORT_MERGE Cost (2)

$$
\begin{aligned}
\mathcal{S}(n,m) &= 3\cdot\mathcal{S}_{size\_t} + \mathcal{S}_{*Quadruple}(n) + \mathcal{S}_{*Quadruple}(m) + \mathcal{S}_{*Quadruple}(n+m) + \\
&\quad + \mathcal{S}_{IntList}(n+m) + \mathcal{S}_{IntList}(n+m+1) + \mathcal{S}_{PairList}(n+m) = \\
&= 3\cdot 8\,\mathrm{B} + n\cdot 16\,\mathrm{B} + m\cdot 16\,\mathrm{B} + (n+m)\cdot 16\,\mathrm{B} + \\
&\quad + ((n+m)\cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + ((n+m+1)\cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + \\
&\quad + ((n+m)\cdot 8\,\mathrm{B} + 16\,\mathrm{B}) = \\
&= 48\,\mathrm{B}\cdot(n+m) + 76\,\mathrm{B} = \\
&= \mathcal{O}(n+m)
\end{aligned}
$$

INSERTIONSERIES_SORT_RECURSIVE Function


```python
def insertionseries_sort_recursive(XY):
    XY = list(XY)
    t = len(XY)
    if t <= 1: return XY
    s = t // 2
    L = insertionseries_sort_recursive(XY[:s])
    R = insertionseries_sort_recursive(XY[s:])
    return insertionseries_sort_merge(L, R)
```

INSERTIONSERIES_SORT_RECURSIVE Function (2)

Algorithm insertionseries_sort_recursive Function

Input:
tuples: the array of tuples to be sort
Output:
result: the sorted array of tuples

function INSERTIONSERIESSORTRECURSIVE(*tuples*)
    if $tuples.size \leq 1$ **then**
        $result \leftarrow tuples$
        return *result*
    end if
    $left \leftarrow [\,]$
    for $i = 0$ **to** $\frac{tuples.size}{2}$ **do**
        $left[i] \leftarrow tuples[i]$
    end for
    $right \leftarrow [\,]$
    for $j = \frac{tuples.size}{2}$ **to** $tuples.size$ **do**
        $right[j] \leftarrow tuples[j]$
    end for
    $leftSorted \leftarrow$ INSERTIONSERIESSORTRECURSIVE(*left*)
    $rightSorted \leftarrow$ INSERTIONSERIESSORTRECURSIVE(*right*)
    $result \leftarrow$ INSERTIONSERIESSORTMERGE(*leftSorted*, *rightSorted*)
    return *result*
end function

INSERTIONSERIES_SORT_RECURSIVE Cost

Let $m := tuples.size$.

$$\mathcal{C}(m) = \begin{cases} \mathcal{C}_{pairlist\_copy}(m) & m \leq 1 \\ 2 \cdot \left(\mathcal{C}_{pairlist\_init}\left(\frac{m}{2}\right) + \mathcal{C}_{pairlist\_reserve}\left(\frac{m}{2}\right)\right) + & \\ \quad + m \cdot \mathcal{C}_{pairlist\_append}\left(\frac{m}{2}\right) + 2 \cdot \mathcal{C}\left(\frac{m}{2}\right) + & \\ \quad + \mathcal{C}_{insertionseries\_sort\_merge}\left(\frac{m}{2}, \frac{m}{2}\right) + & \\ \quad + 4 \cdot \mathcal{C}_{pairlist\_free}\left(\frac{m}{2}\right) & m > 1 \end{cases}$$

$$= \begin{cases} m & m \leq 1 \\ 2 \cdot \left(1 + \frac{m}{2}\right) + m \cdot 1 + 2 \cdot \mathcal{C}\left(\frac{m}{2}\right) + & \\ \quad + \mathcal{O}\left(\left(\frac{m}{2} + \frac{m}{2}\right) \cdot \left(\log\left(\frac{m}{2} + \frac{m}{2}\right)\right)^2\right) + 4 \cdot 1 & m > 1 \end{cases}$$

$$= \begin{cases} 1 & m \leq 1 \\ 2 \cdot \mathcal{C}\left(\frac{m}{2}\right) + 2 \cdot m + 6 + \mathcal{O}\left(m \cdot (\log(m))^2\right) & m > 1 \end{cases}$$

What is currently shown is the time complexity of an iteration; to find the total complexity, the *master theorem* must be applied.

INSERTIONSERIES_SORT_RECURSIVE Cost (2)

$$
\begin{aligned}
\mathcal{C}(m) &= 2 \cdot \mathcal{C}\left(\frac{m}{2}\right) + \mathcal{O}\left(m \cdot (\log(m))^2\right) = \\
&= \mathcal{O}\left(m \cdot (\log(m))^3\right)
\end{aligned}
$$

INSERTIONSERIES_SORT_RECURSIVE Cost (3)

$$\mathcal{S}_{iteration}(m) = \begin{cases} \mathcal{S}_{PairList}(m) & m \leq 1 \\ 2 \cdot \mathcal{S}_{size\_t} + 4 \cdot \mathcal{S}_{PairList}\left(\frac{m}{2}\right) + \mathcal{S}_{PairList}(m) & m > 1 \end{cases}$$

$$= \begin{cases} 8\,\mathrm{B} \cdot m + 16\,\mathrm{B} & m \leq 1 \\ 2 \cdot 8\,\mathrm{B} + 4 \cdot \left(8\,\mathrm{B} \cdot \frac{m}{2} + 16\,\mathrm{B}\right) + (8\,\mathrm{B} \cdot m + 16\,\mathrm{B}) & m > 1 \end{cases}$$

$$= \begin{cases} 24\,\mathrm{B} & m \leq 1 \\ 24\,\mathrm{B} \cdot m + 96\,\mathrm{B} & m > 1 \end{cases}$$

$$\mathcal{S}(m) = \sum_{i=0}^{\log m} \mathcal{S}_{iteration}\left(\frac{m}{2^i}\right) =$$

$$= 24\,\mathrm{B} \cdot m \cdot \sum_{i=0}^{\log m} \frac{1}{2^i} + 96\,\mathrm{B} \cdot \sum_{i=0}^{\log m} 1 =$$

$$< 24\,\mathrm{B} \cdot m \cdot \left(2 - \frac{1}{m}\right) + 96\,\mathrm{B} \cdot (\log m + 1) =$$

$$= 48\,\mathrm{B} \cdot m + 96\,\mathrm{B} \cdot \log(m) + 72\,\mathrm{B}$$

$$= \mathcal{O}(m)$$

INSERTIONSERIES_MERGE_AFTER_SORT_RECURSIVE Function


```python
def insertionseries_merge_after_sort_recursive(L, XY):
    L = list(enumerate(L))
    R = insertionseries_sort_recursive(XY)
    return [y for x, y in insertionseries_sort_merge(L, R)]
```

INSERTIONSERIES_MERGE_AFTER_SORT_RECURSIVE Function (2)

Algorithm insertionseries_merge_after_sort_recursive Function

Input:

array: the array in which new elements will be added

tuples: the array of positions and elements to be added

Output:

result: the array containing the old elements and the new ones inserted in the specified positions

```
function INSERTIONSERIESMERGEAFTERSORTRECURSIVE(array, tuples)
    arrayTuple ← [ ]
    for i = 0 to array.size do
        arrayTuple ← ⟨i, array[i]⟩
    end for
    tuplesSorted ← INSERTIONSERIESSORTRECURSIVE(tuples)
    mergedTuples ← INSERTIONSERIESSORTMERGE(arrayTuple, tuplesSorted)
    result ← [ ]
    for i = 0 to mergedTuples.size do
        ⟨_, value⟩ ← mergedTuples[i]
        result[i] ← value
    end for
    return result
end function
```

INSERTIONSERIES_MERGE_AFTER_SORT_RECURSIVE Cost

Let $n := array.size$ and $m := tuples.size$.

$$\begin{aligned}
\mathcal{C}(n, m) &= \mathcal{C}_{pairlist_init}(n) + \mathcal{C}_{pairlist_reserve}(n) + n \cdot \mathcal{C}_{pairlist_append}(n) + \\
&\quad + \mathcal{C}_{insertionseries_sort_recursive}(m) + \mathcal{C}_{insertionseries_sort_merge}(n, m) + \\
&\quad + \mathcal{C}_{intlist_init}(n+m) + \mathcal{C}_{intlist_reserve}(n+m) + (n+m) \cdot \mathcal{C}_{intlist_append}(m) + \\
&\quad + \mathcal{C}_{pairlist_free}(n) + \mathcal{C}_{pairlist_free}(m) + \mathcal{C}_{pairlist_free}(n+m) = \\
&= 1 + n + n \cdot 1 + \mathcal{O}\left(m \cdot (\log(m))^3\right) + \\
&\quad + \left(8 \cdot (n+m) + \left(\frac{n+m}{2} \cdot \log(n+m) \cdot \frac{\log(n+m)+1}{2}\right) + 12\right) + \\
&\quad + 1 + (n+m) + (n+m) \cdot 1 + 1 + 1 + 1 = \\
&= 2 \cdot n + 10 \cdot (n+m) + \frac{n+m}{2} \cdot \log(n+m) \cdot \frac{\log(n+m)+1}{2} + \\
&\quad + \mathcal{O}\left(m \cdot (\log(m))^3\right) + 17 = \\
&= \mathcal{O}\left((n+m) \cdot (\log(n+m))^2 + m \cdot (\log(m))^3\right)
\end{aligned}$$

INSERTIONSERIES_MERGE_AFTER_SORT_RECURSIVE Cost (2)

$$
\begin{aligned}
\mathcal{S}(n, m) &= \mathcal{S}_{Pairlist}(n) + \mathcal{S}_{Pairlist}(m) + \mathcal{S}_{Pairlist}(n+m) + \mathcal{S}_{IntList}(n+m) = \\
&= (n \cdot 8\,\mathrm{B} + 16\,\mathrm{B}) + (m \cdot 8\,\mathrm{B} + 16\,\mathrm{B}) + \\
&\quad + ((n+m) \cdot 8\,\mathrm{B} + 16\,\mathrm{B}) + ((n+m) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) = \\
&= 20\,\mathrm{B} \cdot (n+m) + 64\,\mathrm{B} = \\
&= \mathcal{O}(n+m)
\end{aligned}
$$

Insertion Series Example

Now let's look at an example that shows how the algorithm works.

- Consider the following array to which the elements will be added:

$$array := \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

- Consider the following array of $\langle position, element \rangle$ pairs that we are going to insert:

$$tuples := \begin{bmatrix} \langle 3, D \rangle & \langle 5, F \rangle & \langle 0, A \rangle & \langle 8, I \rangle \end{bmatrix}$$

The elements are represented as letters to make it easier to follow the algorithm.

Insertion Series Example (2)

Algorithm insertion-series_merge_after_sort_recursive Function

Input:
array: the array in which new elements will be added
tuples: the array of positions and elements to be added

Output:
result: the array containing the old elements and the new ones inserted in the specified positions

```
1  function InsertionSeriesMergeAfterSortRecursive(array, tuples)
2      arrayTuple ← [ ]
3      for i = 0 to array.size do
4          arrayTuple ← ⟨i, array[i]⟩
5      end for
6      tuplesSorted ← InsertionSeriesSortRecursive(tuples)
7      mergedTuples ← InsertionSeriesSortMerge(arrayTuple, tuplesSorted)
8      result ← [ ]
9      for i = 0 to mergedTuples.size do
10         ⟨_, value⟩ ← mergedTuples[i]
11         result[i] ← value
12     end for
13     return result
14 end function
```

- Before *tuplesSorted* (line 5)

$$array = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

$$tuples = \begin{bmatrix} \langle 3, D \rangle & \langle 5, F \rangle & \langle 0, A \rangle & \langle 8, I \rangle \end{bmatrix}$$

Insertion Series Example (3)

Algorithm insertion-series_sort_recursive Function

Input:
tuples: the array of tuples to be sort
Output:
result: the sorted array of tuples

```
1  function INSERTIONSERIESSORTRECURSIVE(tuples)
2      if tuples.size ≤ 1 then
3          result ← tuples
4          return result
5      end if
6      left ← [ ]
7      for i = 0 to tuples.size/2 do
8          left[i] ← tuples[i]
9      end for
10     right ← [ ]
11     for j = tuples.size/2 to tuples.size do
12         right[j] ← tuples[j]
13     end for
14     leftSorted ← INSERTIONSERIESSORTRECURSIVE(left)
15     rightSorted ← INSERTIONSERIESSORTRECURSIVE(right)
16     result ← INSERTIONSERIESSORTMERGE(leftSorted, rightSorted)
17     return result
18 end function
```

- Input (line 1)

$$tuples = [\langle 3, D\rangle \ \langle 5, F\rangle \ \langle 0, A\rangle \ \langle 8, I\rangle]$$

- Before *leftSorted* (line 13)

$$left = \begin{bmatrix} \langle 3, D\rangle & \langle 5, F\rangle \end{bmatrix}$$

Call INSERTIONSERIESSORTRECURSIVE

- Input (line 1)

$$tuples = \begin{bmatrix} \langle 3, D\rangle & \langle 5, F\rangle \end{bmatrix}$$

- Before *leftSorted* (line 13)

$$left = \begin{bmatrix} \langle 3, D\rangle \end{bmatrix}$$

Call INSERTIONSERIESSORTRECURSIVE

- Base case (line 2)

$$tuples = \begin{bmatrix} \langle 3, D\rangle \end{bmatrix}$$

Return

- After *leftSorted* (line 14)

$$leftSorted = \begin{bmatrix} \langle 3, D\rangle \end{bmatrix}$$

- Before *rightSorted* (line 14)

$$right = \begin{bmatrix} \langle 5, F\rangle \end{bmatrix}$$

Call INSERTIONSERIESSORTRECURSIVE

- Base case (line 2)

$$tuples = \begin{bmatrix} \langle 5, F\rangle \end{bmatrix}$$

Return

- After *rightSorted* (line 15)

$$rightSorted = \begin{bmatrix} \langle 5, F\rangle \end{bmatrix}$$

- Before *result* (line 15)

$$leftSorted = \begin{bmatrix} \langle 3, D\rangle \end{bmatrix}$$

$$rightSorted = \begin{bmatrix} \langle 5, F\rangle \end{bmatrix}$$

Call INSERTIONSERIESSORTMERGE

Insertion Series Example (4)

Algorithm insertion-series_sort_merge Function

Input:
tuples1: the first array of tuples that needs to be merged
tuples2: the second array of tuples that needs to be merged
Output:
result: the ordered union of the two input array of tuples

```
1  function INSERTIONSERIESSORTMERGE(tuples1, tuples2)
2      quadruples1 ← [ ]
3      quadruples2 ← [ ]
4      for i = 0 to tuples1.size do
5          ⟨position, element⟩ ← tuples1[i]
6          quadruples1[i] ← ⟨position, element, fromT1, 0⟩
7      end for
8      for j = 0 to tuples2.size do
9          ⟨position, element⟩ ← tuples2[j]
10         quadruples2[i] ← ⟨position − j, element, fromT2, j⟩
11     end for
12     quadruplesMerged ← MERGE(quadruples1, quadruples2)
13     inverses ← [ ]
14     for i = 0 to quadruplesMerged.size do
15         ⟨_, _, fromTuple, _⟩ ← quadruplesMerged[i]
16         inverses[i] ← 1 − fromTuple
17     end for
18     offsets ← PREFIXSUM(inverses)
19     result ← [ ]
20     for i = 0 to quadruplesMerged.size do
21         ⟨key, value, _, _⟩ ← quadruplesMerged[i]
22         result[i] ← ⟨key + offsets[i], value⟩
23     end for
24     return result
25 end function
```

- Input (line 1)

$$tuples1 = \begin{bmatrix} \langle 3, D \rangle \end{bmatrix}$$

$$tuples2 = \begin{bmatrix} \langle 5, F \rangle \end{bmatrix}$$

- Before *quadruplesMerged* (line 11)

$$tuples1 = \begin{bmatrix} \langle 3, D, 1, 0 \rangle \end{bmatrix}$$

$$tuples2 = \begin{bmatrix} \langle 5, F, 0, 0 \rangle \end{bmatrix}$$

Call MERGE

- After *quadruplesMerged* (line 12)

$$quadruplesMerged = [\langle 3, D, 1, 0 \rangle \quad \langle 5, F, 0, 0 \rangle]$$

- After *inverse* (line 17)

$$inverses = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

- Before *offsets* (line 17)

$$inverses = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

Call PREFIXSUM

- After *offsets* (line 18)

$$offsets = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

- After *result* (line 23)

$$result = \begin{bmatrix} \langle 3, D \rangle & \langle 5, F \rangle \end{bmatrix}$$

Insertion Series Example (5)

Algorithm insertion-series_sort_recursive Function

Input:
tuples: the array of tuples to be sort
Output:
result: the sorted array of tuples

1 **function** InsertionSeriesSortRecursive(*tuples*)
2 **if** $tuples.size \leq 1$ **then**
3 $result \leftarrow tuples$
4 **return** *result*
5 **end if**
6 $left \leftarrow [\,]$
7 **for** $i = 0$ **to** $\frac{tuples.size}{2}$ **do**
8 $left[i] \leftarrow tuples[i]$
9 **end for**
10 $right \leftarrow [\,]$
11 **for** $j = \frac{tuples.size}{2}$ **to** $tuples.size$ **do**
12 $right[j] \leftarrow tuples[j]$
13 **end for**
14 $leftSorted \leftarrow$ InsertionSeriesSortRecursive(*left*)
15 $rightSorted \leftarrow$ InsertionSeriesSortRecursive(*right*)
16 $result \leftarrow$ InsertionSeriesSortMerge(*leftSorted*, *rightSorted*)
17 **return** *result*
18 **end function**

- After *result* (line 16)

$$result = \left[\langle 3, D\rangle \quad \langle 5, F\rangle\right]$$

- After *leftSorted* (line 14)

$$leftSorted = \left[\langle 3, D\rangle \quad \langle 5, F\rangle\right]$$

- Before *rightSorted* (line 14)

$$right = \left[\langle 0, A\rangle \quad \langle 8, I\rangle\right]$$

Call InsertionSeries-SortRecursive

- Input (line 1)

$$tuples = \left[\langle 0, A\rangle \quad \langle 8, I\rangle\right]$$

- Before *leftSorted* (line 13)

$$left = \left[\langle 0, A\rangle\right]$$

Call InsertionSeries-SortRecursive

- Base case (line 2)

$$tuples = \left[\langle 0, A\rangle\right]$$

Return

- After *leftSorted* (line 14)

$$leftSorted = \left[\langle 0, A\rangle\right]$$

- Before *rightSorted* (line 14)

$$right = \left[\langle 8, I\rangle\right]$$

Call InsertionSeries-SortRecursive

- Base case (line 2)

$$tuples = \left[\langle 8, I\rangle\right]$$

Return

- After *rightSorted* (line 15)

$$rightSorted = \left[\langle 8, I\rangle\right]$$

- Before *result* (line 15)

$$leftSorted = \left[\langle 0, A\rangle\right]$$

$$rightSorted = \left[\langle 8, I\rangle\right]$$

Call Insertion-SeriesSortMerge

Insertion Series Example (6)

Algorithm insertion-series_sort_merge Function

Input:
*tuples*1: the first array of tuples that needs to be merged
*tuples*2: the second array of tuples that needs to be merged
Output:
result: the ordered union of the two input array of tuples

```
1  function InsertionSeriesSortMerge(tuples1, tuples2)
2      quadruples1 ← [ ]
3      quadruples2 ← [ ]
4      for i = 0 to tuples1.size do
5          ⟨position, element⟩ ← tuples1[i]
6          quadruples1[i] ← ⟨position, element, fromT1, 0⟩
7      end for
8      for j = 0 to tuples2.size do
9          ⟨position, element⟩ ← tuples2[j]
10         quadruples2[i] ← ⟨position − j, element, fromT2, j⟩
11     end for
12     quadruplesMerged ← Merge(quadruples1, quadruples2)
13     inverses ← [ ]
14     for i = 0 to quadruplesMerged.size do
15         ⟨_, _, fromTuple, _⟩ ← quadruplesMerged[i]
16         inverses[i] ← 1 − fromTuple
17     end for
18     offsets ← PrefixSum(inverses)
19     result ← [ ]
20     for i = 0 to quadruplesMerged.size do
21         ⟨key, value, _, _⟩ ← quadruplesMerged[i]
22         result[i] ← ⟨key + offsets[i], value⟩
23     end for
24     return result
25 end function
```

- Input (line 1)

$$tuples1 = \begin{bmatrix} \langle 0, A \rangle \end{bmatrix}$$

$$tuples2 = \begin{bmatrix} \langle 8, I \rangle \end{bmatrix}$$

- Before *quadruplesMerged* (line 11)

$$tuples1 = \begin{bmatrix} \langle 0, A, 1, 0 \rangle \end{bmatrix}$$

$$tuples2 = \begin{bmatrix} \langle 8, I, 0, 0 \rangle \end{bmatrix}$$

Call Merge

- After *quadruplesMerged* (line 12)

$$quadruplesMerged = \begin{bmatrix} \langle 0, A, 1, 0 \rangle & \langle 8, I, 0, 0 \rangle \end{bmatrix}$$

- After *inverses* (line 17)

$$inverses = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

- Before *offsets* (line 17)

$$inverses = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

Call PrefixSum

- After *offsets* (line 18)

$$offsets = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

- After *result* (line 23)

$$result = \begin{bmatrix} \langle 0, A \rangle & \langle 8, I \rangle \end{bmatrix}$$

Insertion Series Example (7)

Algorithm insertion-series_sort_recursive Function

Input:
tuples: the array of tuples to be sort

Output:
result: the sorted array of tuples

```
1  function INSERTIONSERIESSORTRECURSIVE(tuples)
2      if tuples.size ≤ 1 then
3          result ← tuples
4          return result
5      end if
6      left ← [ ]
7      for i = 0 to tuples.size/2 do
8          left[i] ← tuples[i]
9      end for
10     right ← [ ]
11     for j = tuples.size/2 to tuples.size do
12         right[j] ← tuples[j]
13     end for
14     leftSorted ← INSERTIONSERIESSORTRECURSIVE(left)
15     rightSorted ← INSERTIONSERIESSORTRECURSIVE(right)
16     result ← INSERTIONSERIESSORTMERGE(leftSorted, rightSorted)
17     return result
18 end function
```

- After *result* (line 16)

$$result = \begin{bmatrix} \langle 0, A \rangle & \langle 8, I \rangle \end{bmatrix}$$

- After *rightSorted* (line 15)

$$rightSorted = \begin{bmatrix} \langle 0, A \rangle & \langle 8, I \rangle \end{bmatrix}$$

- Before *result* (line 15)

$$leftSorted = \begin{bmatrix} \langle 3, D \rangle & \langle 5, F \rangle \end{bmatrix}$$

$$rightSorted = \begin{bmatrix} \langle 0, A \rangle & \langle 8, I \rangle \end{bmatrix}$$

Call INSERTIONSERIESSORTMERGE

Insertion Series Example (8)

Algorithm insertion-series_sort_merge Function

Input:
tuples1: the first array of tuples that needs to be merged
tuples2: the second array of tuples that needs to be merged
Output:
result: the ordered union of the two input array of tuples

```
1  function InsertionSeriesSortMerge(tuples1, tuples2)
2      quadruples1 ← [ ]
3      quadruples2 ← [ ]
4      for i = 0 to tuples1.size do
5          ⟨position, element⟩ ← tuples1[i]
6          quadruples1[i] ← ⟨position, element, fromT1, 0⟩
7      end for
8      for j = 0 to tuples2.size do
9          ⟨position, element⟩ ← tuples2[j]
10         quadruples2[i] ← ⟨position − j, element, fromT2, j⟩
11     end for
12     quadruplesMerged ← Merge(quadruples1, quadruples2)

13     inverses ← [ ]
14     for i = 0 to quadruplesMerged.size do
15         ⟨_, _, fromTuple, _⟩ ← quadruplesMerged[i]
16         inverses[i] ← 1 − fromTuple
17     end for
18     offsets ← PrefixSum(inverses)
19     result ← [ ]
20     for i = 0 to quadruplesMerged.size do
21         ⟨key, value, _, _⟩ ← quadruplesMerged[i]
22         result[i] ← ⟨key + offsets[i], value⟩
23     end for
24     return result
25 end function
```

- Input (line 1)

$$tuples1 = \begin{bmatrix} \langle 3, D \rangle & \langle 5, F \rangle \end{bmatrix}$$

$$tuples2 = \begin{bmatrix} \langle 0, A \rangle & \langle 8, I \rangle \end{bmatrix}$$

- Before *quadruplesMerged* (line 11)

$$tuples1 = \begin{bmatrix} \langle 3, D, 1, 0 \rangle & \langle 5, F, 1, 0 \rangle \end{bmatrix}$$

$$tuples2 = \begin{bmatrix} \langle 0, A, 0, 0 \rangle & \langle 7, I, 0, 1 \rangle \end{bmatrix}$$

Call Merge

- After *quadruplesMerged* (line 12)

$$quadruplesMerged = [\langle 0, A, 0, 0 \rangle \ \langle 3, D, 1, 0 \rangle \ \langle 5, F, 1, 0 \rangle \ \langle 7, I, 0, 1 \rangle]$$

- After *inverses* (line 17)

$$inverses = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}$$

- Before *offsets* (line 17)

$$inverses = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}$$

Call PrefixSum

- After *offsets* (line 18)

$$offsets = \begin{bmatrix} 0 & 1 & 1 & 1 & 2 \end{bmatrix}$$

- After *result* (line 23)

$$result = [\langle 0, A \rangle \ \langle 4, D \rangle \ \langle 6, F \rangle \ \langle 8, I \rangle]$$

Insertion Series Example (9)

Algorithm insertion-series_sort_recursive Function

Input:
tuples: the array of tuples to be sort
Output:
result: the sorted array of tuples

```
1  function InsertionSeriesSortRecursive(tuples)
2      if tuples.size ≤ 1 then
3          result ← tuples
4          return result
5      end if
6      left ← [ ]
7      for i = 0 to tuples.size/2 do
8          left[i] ← tuples[i]
9      end for
10     right ← [ ]
11     for j = tuples.size/2 to tuples.size do
12         right[j] ← tuples[j]
13     end for
14     leftSorted ← InsertionSeriesSortRecursive(left)
15     rightSorted ← InsertionSeriesSortRecursive(right)
16     result ← InsertionSeriesSortMerge(leftSorted, rightSorted)
17     return result
18 end function
```

- After *result* (line 16)

$$result = \left[\langle 0, A\rangle \quad \langle 4, D\rangle \quad \langle 6, F\rangle \quad \langle 8, I\rangle\right]$$

Insertion Series Example (10)

Algorithm insertion-series_merge_after_sort_recursive Function

Input:
array: the array in which new elements will be added
tuples: the array of positions and elements to be added
Output:
result: the array containing the old elements and the new ones inserted in the specified positions

```
1  function INSERTIONSERIESMERGEAFTERSORTRECURSIVE(array, tuples)
2      arrayTuple ← [ ]
3      for i = 0 to array.size do
4          arrayTuple ← ⟨i, array[i]⟩
5      end for
6      tuplesSorted ← INSERTIONSERIESSORTRECURSIVE(tuples)
7      mergedTuples ← INSERTIONSERIESSORTMERGE(arrayTuple, tuplesSorted)

8      result ← [ ]
9      for i = 0 to mergedTuples.size do
10         ⟨_, value⟩ ← mergedTuples[i]
11         result[i] ← value
12     end for
13     return result
14 end function
```

- After *tuplesSorted* (line 6)

$$array = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$
$$tuplesSorted = \begin{bmatrix} \langle 0, A \rangle & \langle 4, D \rangle & \langle 6, F \rangle & \langle 8, I \rangle \end{bmatrix}$$

- Before *mergedTuples* (line 6)

$$arrayTuple = \begin{bmatrix} \langle 0, 1 \rangle & \langle 1, 2 \rangle & \langle 2, 3 \rangle & \langle 3, 4 \rangle & \langle 4, 5 \rangle \end{bmatrix}$$
$$tuplesSorted = \begin{bmatrix} \langle 0, A \rangle & \langle 4, D \rangle & \langle 6, F \rangle & \langle 8, I \rangle \end{bmatrix}$$

Call INSERTIONSERIESSORTMERGE

Insertion Series Example (11)

Algorithm insertion-series_sort_merge Function

Input:
tuples1: the first array of tuples that needs to be merged
tuples2: the second array of tuples that needs to be merged

Output:
result: the ordered union of the two input array of tuples

```
1  function INSERTIONSERIESSORTMERGE(tuples1, tuples2)
2      quadruples1 ← [ ]
3      quadruples2 ← [ ]
4      for i = 0 to tuples1.size do
5          ⟨position, element⟩ ← tuples1[i]
6          quadruples1[i] ← ⟨position, element, fromT1, 0⟩
7      end for
8      for j = 0 to tuples2.size do
9          ⟨position, element⟩ ← tuples2[j]
10         quadruples2[i] ← ⟨position − j, element, fromT2, j⟩
11     end for
12     quadruplesMerged ← MERGE(quadruples1, quadruples2)
13     inverses ← [ ]
14     for i = 0 to quadruplesMerged.size do
15         ⟨_, _, fromTuple, _⟩ ← quadruplesMerged[i]
16         inverses[i] ← 1 − fromTuple
17     end for
18     offsets ← PREFIXSUM(inverses)
19     result ← [ ]
20     for i = 0 to quadruplesMerged.size do
21         ⟨key, value, _, _⟩ ← quadruplesMerged[i]
22         result[i] ← ⟨key + offsets[i], value⟩
23     end for
24     return result
25 end function
```

- Input (line 1)

$$tuples1 = \begin{bmatrix} \langle 0,1\rangle & \langle 1,2\rangle & \langle 2,3\rangle & \langle 3,4\rangle & \langle 4,5\rangle \end{bmatrix}$$

$$tuples2 = \begin{bmatrix} \langle 0,A\rangle & \langle 4,D\rangle & \langle 6,F\rangle & \langle 8,I\rangle \end{bmatrix}$$

- Before *quadruplesMerged* (line 11)

$$tuples1 = [\langle 0,1,1,0\rangle \ \langle 1,2,1,0\rangle \ \langle 2,3,1,0\rangle \ \langle 3,4,1,0\rangle \ \langle 4,5,1,0\rangle]$$

$$tuples2 = [\langle 0,A,0,0\rangle \ \langle 3,D,0,1\rangle \ \langle 4,F,0,2\rangle \ \langle 5,I,0,3\rangle]$$

Call MERGE

- After *quadruplesMerged* (line 12)

$$quadruplesMerged = [\langle 0,A,0,0\rangle \ \langle 0,1,1,0\rangle \ \langle 1,2,1,0\rangle \ \langle 2,3,1,0\rangle \ \langle 3,D,0,1\rangle \ \langle 3,4,1,0\rangle \ \langle 4,F,0,2\rangle \ \langle 4,5,1,0\rangle \ \langle 5,I,0,3\rangle]$$

- After *inverses* (line 17)

$$inverses = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

- Before *offsets* (line 17)

$$inverses = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Call PREFIXSUM

- After *offsets* (line 18)

$$offsets = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 2 & 2 & 3 & 3 & 4 \end{bmatrix}$$

- After *result* (line 23)

$$result = [\langle 0,A\rangle \ \langle 1,1\rangle \ \langle 2,2\rangle \ \langle 3,3\rangle \ \langle 4,D\rangle \ \langle 5,4\rangle \ \langle 6,F\rangle \ \langle 7,5\rangle \ \langle 8,I\rangle]$$

Insertion Series Example (12)

Algorithm insertion-series_merge_after_sort_recursive Function

Input:
array: the array in which new elements will be added
tuples: the array of positions and elements to be added
Output:
result: the array containing the old elements and the new ones inserted in the specified positions

```
1  function INSERTIONSERIESMERGEAFTERSORTRECURSIVE(array, tuples)
2      arrayTuple ← [ ]
3      for i = 0 to array.size do
4          arrayTuple ← ⟨i, array[i]⟩
5      end for
6      tuplesSorted ← INSERTIONSERIESSORTRECURSIVE(tuples)
7      mergedTuples ← INSERTIONSERIESSORTMERGE(arrayTuple, tuplesSorted)

8      result ← [ ]
9      for i = 0 to mergedTuples.size do
10         ⟨_, value⟩ ← mergedTuples[i]
11         result[i] ← value
12     end for
13     return result
14 end function
```

- After *mergedTuples* (line 7)

$$mergedTuples = [\langle 0, A\rangle \ \langle 1, 1\rangle \ \langle 2, 2\rangle \ \langle 3, 3\rangle \ \langle 4, D\rangle \ \langle 5, 4\rangle \ \langle 6, F\rangle \ \langle 7, 5\rangle \ \langle 8, I\rangle]$$

- After *result* (line 12)

$$result = \begin{bmatrix} A & 1 & 2 & 3 & D & 4 & F & 5 & I \end{bmatrix}$$

Comparison Between Naive and DJB Versions

Let $n :=$ the size of the array to which new elements are added and $m :=$ the size of the array of $\langle position, element\rangle$ to be added.

$$\mathcal{C}_{naive}(n, m) = \mathcal{O}(m \cdot n)$$

$$\mathcal{C}_{DJB}(n, m) = \mathcal{O}\Big((n + m) \cdot (\log{(n + m)})^2 + m \cdot (\log{(m)})^3\Big)$$

Thus for large sizes of n, m the algorithm devised by DJB speeds up the insertion of multiples of elements into an array at the same time.
Obviously if you only want to insert only a single element, the best algorithm is the naive one which has $\mathcal{O}(n)$ complexity and not DJB which has $\mathcal{O}\Big(n \cdot (\log{(n)})^2\Big)$ instead.

Using Parallelisation

To try to gain computation time, the technique of parallelization can be used.
In this algorithm, the functions that can be parallelized are essentially MERGE and PREFIXSUM.

- MERGE: it was designed to be easily parallelised. The *bitonic sort* network was chosen to sort the data, an algorithm particularly suited to parallelisation due to its regular and predetermined structure.
- PREFIXSUM: the parallel version must be modified slightly as it is not suitable for parallelization as it stands. Therefore, the parallel version divides the input into blocks managed by separate threads, which compute the partial sums of their segments in parallel. Next, each thread updates its results by adding the offsets computed from the sums of the previous threads.

Constant-Time

The algorithm was implemented using a constant-time approach.

All data-dependent branches were eliminated and replaced with mux bitmasks for conditional selection. All comparison and swap operations are branchless.
This approach allows us to minimise the possibility of side-channel attacks, as there are no data-dependent branches.

Application of Insertion Series

One application of the INSERTIONSERIES algorithm is to map a sequence of integers into a constant-weight binary word.

Let us now go on to study this application specifically.

Constant-Weight Binary Word

Definition (Constant-Weight Binary Word)

A **Constant-Weight Binary Word** is a set of binary vectors, *codewords*, of the same length and with the same Hamming weight[a].

[a] The Hamming weight of a vector is defined as $\mathrm{HW}(\mathbf{x}) := |\{i \mid x_i \neq 0\}|$

The linear codes are used in Code-Based Cryptography.
The algorithm detailed below can be automated to generate different binary words with the same Hamming weight simply by changing the positions of the 1s to be inserted.

Different Implementation

Daniel J. Bernstein implemented two different algorithms to construct a constant-weight word.

- cww_via_insertionseries;
- cww_merge_after_sort_recursive.

In the following sections, we will analyse them individually.

CWW_VIA_INSERTIONSERIES Description

This solution only uses the INSERTIONSERIES algorithm defined previously without any changes.

CWW_VIA_INSERTIONSERIES Function


```python
def cww_via_insertionseries(m,X):
    return insertionseries([0]*m,((x,1) for x in X))
```

CWW_VIA_INSERTIONSERIES Function (2)

Algorithm cww_via_insertionseries Function

Input:

numberZero: the number of 0s in the constant-weight word
onePositions: the positions in which to insert 1

Output:

result: the constant-weight word

```
function CWWVIAINSERTIONSERIES(numberZero, onePositions)
    zeroPositions ← [ ]
    for i = 0 to numberZero do
        zeroPositions[i] ← i
    end for
    oneTuples ← [ ]
    for i = 0 to onePositions.size do
        oneTuples[i] ← ⟨onePositions[i], 1⟩
    end for
    result ← INSERTIONSERIESMERGEAFTERSORTRECURSIVE(zeroPositions, oneTuples)
    return result
end function
```

CWW_VIA_INSERTIONSERIES Cost

Let $n :=$ *numberZero* and $m :=$ *onePositions.size*.

$$
\begin{aligned}
\mathcal{C}(n, m) &= \mathcal{C}_{intlist\_init}(n) + \mathcal{C}_{intlist\_reserve}(n) + n \cdot \mathcal{C}_{intlist\_append}(n) + \\
&\quad + \mathcal{C}_{pairlist\_init}(m) + \mathcal{C}_{pairlist\_reserve}(m) + m \cdot \mathcal{C}_{pairlist\_append}(m) + \\
&\quad + \mathcal{C}_{insertionseries\_merge\_after\_sort\_recursive}(n, m) + \\
&\quad + \mathcal{C}_{intlist\_free}(n) + \mathcal{C}_{pairlist\_free}(m) = \\
&= 1 + n + n \cdot 1 + 1 + m + m \cdot 1 + \\
&\quad + \left(2 \cdot n + 10 \cdot (n + m) + \frac{n + m}{2} \cdot \log(n + m) \cdot \frac{\log(n + m) + 1}{2} + \mathcal{O}\left(m \cdot (\log(m))^3\right) + 17\right) + \\
&\quad + 1 + 1 = \\
&= 2 \cdot n + 12 \cdot (n + m) + \frac{n + m}{2} \cdot \log(n + m) \cdot \frac{\log(n + m) + 1}{2} + 21 + \mathcal{O}\left(m \cdot (\log(m))^3\right) = \\
&= \mathcal{O}\left((n + m) \cdot (\log(n + m))^2 + m \cdot (\log(m))^3\right)
\end{aligned}
$$

CWW_VIA_INSERTIONSERIES Cost (2)

$$
\begin{aligned}
\mathcal{S}(n, m) &= \mathcal{S}_{IntList}(n) + \mathcal{S}_{PairList}(m) + \mathcal{S}_{IntList}(n+m) = \\
&= (n \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + (m \cdot 8\,\mathrm{B} + 16\,\mathrm{B}) + ((n+m) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) = \\
&= 4\,\mathrm{B} \cdot m + 8\,\mathrm{B} \cdot (n+m) + 48\,\mathrm{B} = \\
&= \mathcal{O}(n+m)
\end{aligned}
$$

Ad-Hoc cww_algorithm

This algorithm represents an optimization of the previous one and, as such, also uses the MERGE and PREFIXSUM functions. These functions will no longer be described explicitly, as they are considered implicit and have not been modified from their previous definition.

Now, we will analyse all the complexities of the auxiliary functions until we reach the actual complexity of the algorithm for creating a constant-weight word.

CWW_SORT_MERGEBITS Function


```python
def cww_sort_mergebits(L,R):
    L = [(x,1) for x in L]
    R = [(x-j,0) for j,x in enumerate(R)]
    M = merge(L,R)
    return [1-fromL for _,fromL in M]
```

CWW_VIA_INSERTIONSERIES Function (3)

Algorithm cww_sort_mergebits Function

Input:

zeroPositions: the positions of the 0s

onePositions: the positions in which to insert 1

Output:

result: the constant-weight word

```
function cwwSortMergebits(zeroPositions, onePositions)
    zeroQuadruples ← [ ]
    for i = 0 to zeroPositions.size do
        zeroQuadruples[i] ← ⟨zeroPositions[i], 0, 1, i⟩
    end for
    oneQuadruples ← [ ]
    for j = 0 to onePositions.size do
        oneQuadruples[j] ← ⟨onePositions[j] − j, 1, 0, j⟩
    end for
    quadruplesMerged ← merge(zeroQuadruples, oneQuadruples)
    result ← [ ]
    for i = 0 to quadruplesMerged.size do
        ⟨_, bit, _, _⟩ ← quadruplesMerged[i⟩]
        result[i] ← bit
    end for
    return result
end function
```

CWW_SORT_MERGEBITS Cost

Let $n := zeroPositions.size$ and $m := onePositions.size$.

$$\begin{aligned}
\mathcal{C}(n,m) &= \mathcal{C}_{malloc}(n) + \mathcal{C}_{malloc}(m) + n + m + \mathcal{C}_{merge}(n,m) + \mathcal{C}_{intlist_init}(n+m) + \\
&\quad + \mathcal{C}_{intlist_reserve}(n+m) + (n+m) \cdot \mathcal{C}_{intlist_append}(n+m) + \\
&\quad + \mathcal{C}_{free}(n) + \mathcal{C}_{free}(m) + \mathcal{C}_{free}(n+m) = \\
&= 1 + 1 + n + m + \left(1 + n + m + \left(\frac{n+m}{2} \cdot \log(n+m) \cdot \frac{\log(n+m)+1}{2}\right)\right) + \\
&\quad + 1 + (n+m) + (n+m) \cdot 1 + 1 + 1 + 1 = \\
&= 4 \cdot (n+m) + \frac{n+m}{2} \cdot \log(n+m) \cdot \frac{\log(n+m)+1}{2} + 7 = \\
&= \mathcal{O}\left((n+m) \cdot (\log(n+m))^2\right)
\end{aligned}$$

CWW_SORT_MERGEBITS Cost (2)

$$
\begin{aligned}
\mathcal{S}(n, m) &= 3 \cdot \mathcal{S}_{size\_t} + \mathcal{S}_{*Quadruple}(n) + \mathcal{S}_{*Quadruple}(m) + \mathcal{S}_{*Quadruple}(n+m) + \\
&\quad + \mathcal{S}_{IntList}(n+m) = \\
&= 3 \cdot 8\,\mathrm{B} + (n \cdot 16\,\mathrm{B}) + (m \cdot 16\,\mathrm{B}) + ((n+m) \cdot 16\,\mathrm{B}) + \\
&\quad + ((n+m) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) = \\
&= 36\,\mathrm{B} \cdot (n+m) + 40\,\mathrm{B} = \\
&= \mathcal{O}(n+m)
\end{aligned}
$$

CWW_SORT_MERGEPOS Function


```python
def cww_sort_mergepos(L,R):
    L = [(x,1) for x in L]
    R = [(x-j,0) for j,x in enumerate(R)]
    M = merge(L,R)
    offsets = prefixsums(1-fromL for _,fromL in M)
    return [x+offset for (x,_),offset in zip(M,offsets)]
```

CWW_SORT_MERGEPOS Function (2)

Algorithm cww_sort_mergepos Function

Input:
*array*1: the first array that needs to be merged
*array*2: the second array that needs to be merged
Output:
result: the ordered union of the two input array

```
function cwwSortMergePos(array1, array2)
    quadruples1 ← [ ]
    for i = 0 to array1.size do
        quadruples1 ← ⟨array1[i], 0, fromA1, 0⟩
    end for
    quadruples2 ← [ ]
    for j = 0 to array2.size do
        quadruples2[j] ← ⟨array2[j] − j, 1, fromA2, j⟩
    end for
    quadruplesMerged ← MERGE(quadruples1, quadruples2)
    inverses ← [ ]
    for i = 0 to quadruplesMerged.size do
        ⟨_, _, fromArray, _⟩ ← quadruplesMerged[i]
        inverses[i] ← 1 − fromArray
    end for
    offsets ← PREFIXSUM(inverses)
    result ← [ ]
    for i = 0 to quadruplesMerged.size do
        ⟨position, _, _, _⟩ ← quadruplesMerged[i]
        result[i] ← position + offsets[i]
    end for
    return result
end function
```

CWW_SORT_MERGEPOS Cost

Let $n := array1.size$ and $m := array2.size$.

$$\begin{aligned}
\mathcal{C}(n,m) &= \mathcal{C}_{malloc}(n) + \mathcal{C}_{malloc}(m) + n + m + \mathcal{C}_{merge}(n,m) + 2 \cdot \mathcal{C}_{intlist_init}(n+m) + \\
&+ 2 \cdot \mathcal{C}_{intlist_reserve}(n+m) + 2 \cdot (n+m) \cdot \mathcal{C}_{intlist_append}(n+m) + \\
&+ \mathcal{C}_{prefixSum}(n+m) + \mathcal{C}_{free}(n) + \mathcal{C}_{free}(m) + \mathcal{C}_{free}(n+m) + \\
&+ \mathcal{C}_{intlist_free}(n+m) + \mathcal{C}_{intlist_free}(n+m+1) = \\
&= 1 + 1 + n + m + \left(1 + n + m + \left(\frac{n+m}{2} \cdot \log(n+m) \cdot \frac{\log(n+m)+1}{2}\right)\right) + \\
&+ 2 \cdot 1 + 2 \cdot (n+m) + 2 \cdot (n+m) \cdot 1 + (2 \cdot n + 2) + 1 + 1 + 1 + 1 + 1 = \\
&= 2 \cdot n + 6 \cdot (n+m) + \frac{n+m}{2} \cdot \log(n+m) \cdot \frac{\log(n+m)+1}{2} + 12 = \\
&= \mathcal{O}\left((n+m) \cdot (\log(n+m))^2\right)
\end{aligned}$$

CWW_SORT_MERGEPOS Cost (2)

$$
\begin{aligned}
\mathcal{S}(n, m) &= 3 \cdot \mathcal{S}_{size\_t} + \mathcal{S}_{*Quadruple}(n) + \mathcal{S}_{*Quadruple}(m) + \mathcal{S}_{*Quadruple}(n + m) + \\
&\quad + 2 \cdot \mathcal{S}_{IntList}(n + m) + \mathcal{S}_{IntList}(n + m + 1) = \\
&= 3 \cdot 8\,\mathrm{B} + n \cdot 16\,\mathrm{B} + m \cdot 16\,\mathrm{B} + (n + m) \cdot 16\,\mathrm{B} + \\
&\quad + 2 \cdot ((n + m) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + ((n + m + 1) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) = \\
&= 44\,\mathrm{B} \cdot (n + m) + 76\,\mathrm{B} = \\
&= \mathcal{O}(n + m)
\end{aligned}
$$

CWW_SORT_RECURSIVE Function


```
def cww_sort_recursive(X):
    X = list(X)
    t = len(X)
    if t <= 1: return X
    s = t//2
    L = cww_sort_recursive(X[:s])
    R = cww_sort_recursive(X[s:])
    return cww_sort_mergepos(L,R)
```

CWW_SORT_RECURSIVE Function (2)

Algorithm cww_sort_recursive Function

Input:
array: the array to be sort
Output:
result: the sorted array

```
function cwwSortRecursive(array)
    if array.size ≤ 1 then
        result ← array
        return result
    end if
    left ← [ ]
    for i = 0 to array.size/2 do
        left[i] ← array[i]
    end for
    right ← [ ]
    for j = array.size/2 to array.size do
        right[j] ← array[j]
    end for
    leftSorted ← cwwSortRecursive(left)
    rightSorted ← cwwSortRecursive(right)
    result ← cwwSortMergePos(leftSorted, rightSorted)
    return result
end function
```

CWW_SORT_RECURSIVE Cost

Let $n := array.size$.

$$\mathcal{C}(n) = \begin{cases} \mathcal{C}_{intlist\_copy}(n) & n \leq 1 \\ 2 \cdot \left(\mathcal{C}_{intlist\_init}\left(\frac{n}{2}\right) + \mathcal{C}_{intlist\_reserve}\left(\frac{n}{2}\right)\right) + n \cdot \mathcal{C}_{intlist\_append}\left(\frac{n}{2}\right) + \\ \quad +2 \cdot \mathcal{C}\left(\frac{n}{2}\right) + \mathcal{C}_{cww\_sort\_mergepos}\left(\frac{n}{2}, \frac{n}{2}\right) + 4 \cdot \mathcal{C}_{intlist\_free}\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

$$= \begin{cases} n & n \leq 1 \\ 2 \cdot \left(1 + \frac{n}{2}\right) + n \cdot 1 + 2 \cdot \mathcal{C}\left(\frac{n}{2}\right) + \\ \quad +\mathcal{O}\left(\left(\frac{n}{2} + \frac{n}{2}\right) \cdot \left(\log\left(\frac{n}{2} + \frac{n}{2}\right)\right)^2\right) + 4 \cdot 1 & n > 1 \end{cases}$$

$$= \begin{cases} 1 & n \leq 1 \\ 2 \cdot \mathcal{C}\left(\frac{n}{2}\right) + 2 \cdot n + 6 + \mathcal{O}\left(n \cdot (\log(n))^2\right) & n > 1 \end{cases}$$

What is currently shown is the time complexity of an iteration; to find the total complexity, the *master theorem* must be applied.

CWW_SORT_RECURSIVE Cost (2)

$$\mathcal{C}(n) = 2 \cdot \mathcal{C}\left(\frac{n}{2}\right) + \mathcal{O}\left(n \cdot (\log(n))^2\right) =$$
$$= \mathcal{O}\left(n \cdot (\log(n))^3\right)$$

CWW_SORT_RECURSIVE Cost (3)

$$\mathcal{S}_{iteration}(n) = \begin{cases} \mathcal{S}_{IntList}(n) & n \leq 1 \\ 2 \cdot \mathcal{S}_{size\_t} + 4 \cdot \mathcal{S}_{IntList}\left(\frac{n}{2}\right) + \mathcal{S}_{IntList}(n) & n > 1 \end{cases}$$

$$= \begin{cases} 4\,\mathrm{B} \cdot n + 16\,\mathrm{B} & n \leq 1 \\ 2 \cdot 8\,\mathrm{B} + 4 \cdot \left(\frac{n}{2} \cdot 4\,\mathrm{B} + 16\,\mathrm{B}\right) + (n \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) & n > 1 \end{cases}$$

$$= \begin{cases} 20\,\mathrm{B} & n \leq 1 \\ 12\,\mathrm{B} \cdot n + 96\,\mathrm{B} & n > 1 \end{cases}$$

$$\mathcal{S}(n) = \sum_{i=0}^{\log n} \mathcal{S}_{iteration}\left(\frac{n}{2^i}\right) =$$

$$= 12\,\mathrm{B} \cdot n \cdot \sum_{i=0}^{\log n} \frac{1}{2^i} + 96\,\mathrm{B} \cdot \sum_{i=0}^{\log n} 1 =$$

$$< 12\,\mathrm{B} \cdot n \cdot \left(2 - \frac{1}{n}\right) + 96\,\mathrm{B} \cdot (\log n + 1) =$$

$$= 24\,\mathrm{B} \cdot n + 96\,\mathrm{B} \cdot \log n + 84\,\mathrm{B}$$

$$= \mathcal{O}(n)$$

CWW_MERGE_AFTER_SORT_RECURSIVE Function


```
def cww_merge_after_sort_recursive(m,X):
    return cww_sort_mergebits(range(m),cww_sort_recursive(X))
```

CWW_MERGE_AFTER_SORT_RECURSIVE Function (2)

Algorithm cww_merge_after_sort_recursive Function

Input:

numberZero: the number of 0s in the constant-weight word
onePositions: the positions in which to insert 1

Output:

result: the constant-weight word

```
function cwwMergeAfterSortRecursive(numberZero, onePositions)
    zeroPositions ← [ ]
    for i = 0 to numberZero do
        zeroPositions[i] ← i
    end for
    onePositionSorted ← cwwSortRecursive(onePositions)
    result ← cwwSortMergeBits(zeroPositions, onePositionSorted)
    return result
end function
```

CWW_MERGE_AFTER_SORT_RECURSIVE Cost

Let $n :=$ *numberZero* and $m :=$ *onePositions.size*.

$$\begin{aligned}
\mathcal{C}(n, m) &= \mathcal{C}_{intlist_init}(n) + \mathcal{C}_{intlist_reserve}(n) + n \cdot \mathcal{C}_{intlist_append}(n) + \\
&\quad + \mathcal{C}_{cww_sort_recursive}(m) + \mathcal{C}_{cww_sort_mergebits}(n, m) + \\
&\quad + \mathcal{C}_{intlist_free}(n) + \mathcal{C}_{intlist_free}(m) = \\
&= 1 + n + n \cdot 1 + \mathcal{O}\left(m \cdot (\log(m))^3\right) + \\
&\quad + \left(4 \cdot (n + m) + \frac{n + m}{2} \cdot \log(n + m) \cdot \frac{\log(n + m) + 1}{2} + 7\right) + \\
&\quad + 1 + 1 = \\
&= 2 \cdot n + 4 \cdot (n + m) + \frac{n + m}{2} \cdot \log(n + m) \cdot \frac{\log(n + m) + 1}{2} + 10 + \\
&\quad + \mathcal{O}\left(m \cdot (\log(m))^3\right) = \\
&= \mathcal{O}\left((n + m) \cdot (\log(n + m))^2 + m \cdot (\log(m))^3\right)
\end{aligned}$$

CWW_MERGE_AFTER_SORT_RECURSIVE Cost (2)

$$
\begin{aligned}
\mathcal{S}(n, m) &= \mathcal{S}_{IntList}(n) + \mathcal{S}_{IntList}(m) + \mathcal{S}_{IntList}(n+m) = \\
&= (n \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + (m \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + ((n+m) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) = \\
&= 8\,\mathrm{B} \cdot (n+m) + 48\,\mathrm{B} = \\
&= \mathcal{O}(n+m)
\end{aligned}
$$

Ad-Hoc cww_algorithm Example

Now let's look at an example that shows how the algorithm works.

- Consider the following number of 0s in the constant-weight word:

$$numberZero := 5$$

- Consider the following array of positions in which to add a 1:

$$onePositions := \begin{bmatrix} 3 & 5 \end{bmatrix}$$

Ad-Hoc cww_algorithm Example (2)

Algorithm
cww_merge_after_sort_recursive Function

Input:

numberZero: the number of 0s in the constant-weight word

onePositions: the positions in which to insert 1

Output:

result: the constant-weight word

```
1 function cwwMergeAfterSortRecursive(numberZero, onePositions)
2     zeroPositions ← [ ]
3     for i = 0 to numberZero do
4         zeroPositions[i] ← i
5     end for
6     onePositionSorted ← cwwSortRecursive(onePositions)
7     result ← cwwSortMergeBits(zeroPositions, onePositionSorted)
8     return result
9 end function
```

- Before *onePositionSorted* (line 5)

$$numberZero = 5$$

$$onePositions = \begin{bmatrix} 3 & 5 & 0 & 8 \end{bmatrix}$$

Ad-Hoc cww_algorithm Example (3)

Algorithm cww_sort_recursive Function

Input:
array: the array to be sort
Output:
result: the sorted array

```
1  function CWWSORTRECURSIVE(array)
2      if array.size ≤ 1 then
3          result ← array
4          return result
5      end if
6      left ← [ ]
7      for i = 0 to array.size/2 do
8          left[i] ← array[i]
9      end for
10     right ← [ ]
11     for j = array.size/2 to array.size do
12         right[j] ← array[j]
13     end for
14     leftSorted ← CWWSORTRECURSIVE(left)
15     rightSorted ← CWWSORTRECURSIVE(right)
16     result ← CWWSORTMERGEPOS(leftSorted, rightSorted)
17     return result
18 end function
```

- Input (line 1)

$$array = \begin{bmatrix} 3 & 5 & 0 & 8 \end{bmatrix}$$

- Before *leftSorted* (line 13)

$$left = \begin{bmatrix} 3 & 5 \end{bmatrix}$$

Call CWWSORTRECURSIVE

- Input (line 1)

$$array = \begin{bmatrix} 3 & 5 \end{bmatrix}$$

- Before *leftSorted* (line 13)

$$left = \begin{bmatrix} 3 \end{bmatrix}$$

Call CWWSORTRECURSIVE

- Base case (line 2)

$$array = \begin{bmatrix} 3 \end{bmatrix}$$

Return

- After *leftSorted* (line 14)

$$leftSorted = \begin{bmatrix} 3 \end{bmatrix}$$

- Before *rightSorted* (line 14)

$$right = \begin{bmatrix} 5 \end{bmatrix}$$

Call CWWSORTRECURSIVE

- Base case (line 2)

$$array = \begin{bmatrix} 5 \end{bmatrix}$$

Return

- After *rightSorted* (line 15)

$$rightSorted = \begin{bmatrix} 5 \end{bmatrix}$$

- Before *result* (line 15)

$$leftSorted = \begin{bmatrix} 3 \end{bmatrix}$$

$$rightSorted = \begin{bmatrix} 5 \end{bmatrix}$$

Call CWWSORTMERGEPOS

Ad-Hoc cww_algorithm Example (4)

Algorithm cww_sort_mergepos Function

Input:
array1: the first array that needs to be merged
array2: the second array that needs to be merged

Output:
result: the ordered union of the two input array

```
1  function cwwSortMergePos(array1, array2)
2      quadruples1 ← [ ]
3      for i = 0 to array1.size do
4          quadruples1 ← ⟨array1[i], 0, fromA1, 0⟩
5      end for
6      quadruples2 ← [ ]
7      for j = 0 to array2.size do
8          quadruples2[j] ← ⟨array2[j] − j, 1, fromA2, j⟩
9      end for
10     quadruplesMerged ← MERGE(quadruples1, quadruples2)
11     inverses ← [ ]
12     for i = 0 to quadruplesMerged.size do
13         ⟨_, _, fromArray, _⟩ ← quadruplesMerged[i]
14         inverses[i] ← 1 − fromArray
15     end for
16     offsets ← PREFIXSUM(inverses)
17     result ← [ ]
18     for i = 0 to quadruplesMerged.size do
19         ⟨position, _, _, _⟩ ← quadruplesMerged[i]
20         result[i] ← position + offsets[i]
21     end for
22     return result
23 end function
```

- Input (line 1)

$$array1 = \begin{bmatrix} 3 \end{bmatrix}$$

$$array2 = \begin{bmatrix} 5 \end{bmatrix}$$

- Before *quadruplesMerged* (line 9)

$$quadruples1 = \begin{bmatrix} \langle 3, 1, 1, 0 \rangle \end{bmatrix}$$

$$quadruples2 = \begin{bmatrix} \langle 5, 1, 0, 0 \rangle \end{bmatrix}$$

- After *quadruplesMerged* (line 10)

$$quadruplesMerged = \begin{bmatrix} \langle 3, 1, 1, 0 \rangle & \langle 5, 1, 0, 0 \rangle \end{bmatrix}$$

- From *inverses* (line 15)

$$inverses = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

- Before *offsets* (line 15)

$$inverses = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

- After *offsets* (line 16)

$$offsets = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

- After *result* (line 21)

$$result = \begin{bmatrix} 3 & 5 \end{bmatrix}$$

Ad-Hoc cww_algorithm Example (5)

Algorithm cww_sort_recursive Function

Input:
array: the array to be sort
Output:
result: the sorted array

```
1  function cwwSortRecursive(array)
2      if array.size ≤ 1 then
3          result ← array
4          return result
5      end if
6      left ← [ ]
7      for i = 0 to array.size/2 do
8          left[i] ← array[i]
9      end for
10     right ← [ ]
11     for j = array.size/2 to array.size do
12         right[j] ← array[j]
13     end for
14     leftSorted ← cwwSortRecursive(left)
15     rightSorted ← cwwSortRecursive(right)
16     result ← cwwSortMergePos(leftSorted, rightSorted)
17     return result
18 end function
```

- After *result* (line 21)

$$result = \begin{bmatrix} 3 & 5 \end{bmatrix}$$

- After *leftSorted* (line 14)

$$leftSorted = \begin{bmatrix} 3 & 5 \end{bmatrix}$$

- Before *rightSorted* (line 14)

$$right = \begin{bmatrix} 0 & 8 \end{bmatrix}$$

Call cwwSortRecursive

- Input (line 1)

$$array = \begin{bmatrix} 0 & 8 \end{bmatrix}$$

- Before *leftSorted* (line 13)

$$left = \begin{bmatrix} 0 \end{bmatrix}$$

Call cwwSortRecursive

- Base case (line 2)

$$array = \begin{bmatrix} 0 \end{bmatrix}$$

Return

- After *leftSorted* (line 14)

$$leftsorted = \begin{bmatrix} 0 \end{bmatrix}$$

- Before *rightSorted* (line 14)

$$irght = \begin{bmatrix} 8 \end{bmatrix}$$

Call cwwSortRecursive

- Base case (line 2)

$$array = \begin{bmatrix} 8 \end{bmatrix}$$

Return

- After *rightSorted* (line 15)

$$rightSorted = \begin{bmatrix} 8 \end{bmatrix}$$

- Before *result* (line 15)

$$leftSorted = \begin{bmatrix} 0 \end{bmatrix}$$

$$rightSorted = \begin{bmatrix} 8 \end{bmatrix}$$

Call cwwSortMergePos

Ad-Hoc cww_algorithm Example (6)

Algorithm cww_sort_mergepos Function

Input:
*array*1: the first array that needs to be merged
*array*2: the second array that needs to be merged
Output:
result: the ordered union of the two input array

```
 1 function cwwSortMergePos(array1, array2)
 2     quadruples1 ← [ ]
 3     for i = 0 to array1.size do
 4         quadruples1 ← ⟨array1[i], 0, fromA1, 0⟩
 5     end for
 6     quadruples2 ← [ ]
 7     for j = 0 to array2.size do
 8         quadruples2[j] ← ⟨array2[j] − j, 1, fromA2, j⟩
 9     end for
10     quadruplesMerged ← merge(quadruples1, quadruples2)
11     inverses ← [ ]
12     for i = 0 to quadruplesMerged.size do
13         ⟨_, _, fromArray, _⟩ ← quadruplesMerged[i]
14         inverses[i] ← 1 − fromArray
15     end for
16     offsets ← prefixSum(inverses)
17     result ← [ ]
18     for i = 0 to quadruplesMerged.size do
19         ⟨position, _, _, _⟩ ← quadruplesMerged[i]
20         result[i] ← position + offsets[i]
21     end for
22     return result
23 end function
```

- Input (line 1)

$$array1 = \begin{bmatrix} 0 \end{bmatrix}$$

$$array2 = \begin{bmatrix} 8 \end{bmatrix}$$

- Before *quadruplesMerged* (line 9)

$$quadruples1 = \begin{bmatrix} \langle 0, 1, 1, 0 \rangle \end{bmatrix}$$

$$quadruples2 = \begin{bmatrix} \langle 8, 1, 0, 0 \rangle \end{bmatrix}$$

- After *quadruplesMerged* (line 10)

$$quadruplesMerged = \begin{bmatrix} \langle 0, 1, 1, 0 \rangle & \langle 8, 1, 0, 0 \rangle \end{bmatrix}$$

- From *inverses* (line 15)

$$inverses = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

- Before *offsets* (line 15)

$$inverses = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

- After *offsets* (line 16)

$$offsets = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

- After *result* (line 21)

$$result = \begin{bmatrix} 0 & 8 \end{bmatrix}$$

Ad-Hoc cww_algorithm Example (7)

Algorithm cww_sort_recursive Function

Input:
array: the array to be sort
Output:
result: the sorted array

```
1  function cwwSortRecursive(array)
2      if array.size ≤ 1 then
3          result ← array
4          return result
5      end if
6      left ← [ ]
7      for i = 0 to array.size/2 do
8          left[i] ← array[i]
9      end for
10     right ← [ ]
11     for j = array.size/2 to array.size do
12         right[j] ← array[j]
13     end for
14     leftSorted ← cwwSortRecursive(left)
15     rightSorted ← cwwSortRecursive(right)
16     result ← cwwSortMergePos(leftSorted, rightSorted)
17     return result
18 end function
```

- After *result* (line 21)

$$result = \begin{bmatrix} 0 & 8 \end{bmatrix}$$

- After *rightSorted* (line 15)

$$rightSorted = \begin{bmatrix} 0 & 8 \end{bmatrix}$$

- Before *result* (line 15)

$$leftSorted = \begin{bmatrix} 3 & 5 \end{bmatrix}$$
$$rightSorted = \begin{bmatrix} 0 & 8 \end{bmatrix}$$

Call cwwSortMergePos

Ad-Hoc cww_algorithm Example (8)

Algorithm cww_sort_mergepos Function

Input:
*array*1: the first array that needs to be merged
*array*2: the second array that needs to be merged
Output:
result: the ordered union of the two input array

```
1  function CWWSORTMERGEPOS(array1, array2)
2      quadruples1 ← [ ]
3      for i = 0 to array1.size do
4          quadruples1 ← ⟨array1[i], 0, fromA1, 0⟩
5      end for
6      quadruples2 ← [ ]
7      for j = 0 to array2.size do
8          quadruples2[j] ← ⟨array2[j] − j, 1, fromA2, j⟩
9      end for
10     quadruplesMerged ← MERGE(quadruples1, quadruples2)
11     inverses ← [ ]
12     for i = 0 to quadruplesMerged.size do
13         ⟨_, _, fromArray, _⟩ ← quadruplesMerged[i]
14         inverses[i] ← 1 − fromArray
15     end for
16     offsets ← PREFIXSUM(inverses)
17     result ← [ ]
18     for i = 0 to quadruplesMerged.size do
19         ⟨position, _, _, _⟩ ← quadruplesMerged[i]
20         result[i] ← position + offsets[i]
21     end for
22     return result
23 end function
```

- Input (line 1)

$$array1 = \begin{bmatrix} 3 & 5 \end{bmatrix}$$

$$array2 = \begin{bmatrix} 0 & 8 \end{bmatrix}$$

- Before *quadruplesMerged* (line 9)

$$quadruples1 = \begin{bmatrix} \langle 3, 1, 1, 0 \rangle & \langle 5, 1, 1, 1 \rangle \end{bmatrix}$$

$$quadruples2 = \begin{bmatrix} \langle 0, 1, 0, 0 \rangle & \langle 7, 1, 0, 1 \rangle \end{bmatrix}$$

- After *quadruplesMerged* (line 10)

$$quadruplesMerged = [\langle 0, 1, 0, 0 \rangle \ \langle 3, 1, 1, 0 \rangle \ \langle 5, 1, 1, 1 \rangle \ \langle 7, 1, 0, 1 \rangle]$$

- From *inverses* (line 15)

$$inverses = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}$$

- Before *offsets* (line 15)

$$inverses = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}$$

- After *offsets* (line 16)

$$offsets = \begin{bmatrix} 0 & 1 & 1 & 1 & 2 \end{bmatrix}$$

- After *result* (line 21)

$$result = \begin{bmatrix} 0 & 4 & 6 & 8 \end{bmatrix}$$

Ad-Hoc cww_algorithm Example (9)

Algorithm cww_sort_recursive Function

Input:
array: the array to be sort
Output:
result: the sorted array

```
1  function cwwSortRecursive(array)
2      if array.size ≤ 1 then
3          result ← array
4          return result
5      end if
6      left ← [ ]
7      for i = 0 to array.size/2 do
8          left[i] ← array[i]
9      end for
10     right ← [ ]
11     for j = array.size/2 to array.size do
12         right[j] ← array[j]
13     end for
14     leftSorted ← cwwSortRecursive(left)
15     rightSorted ← cwwSortRecursive(right)
16     result ← cwwSortMergePos(leftSorted, rightSorted)
17     return result
18 end function
```

- After *result* (line 21)

$$result = \begin{bmatrix} 0 & 4 & 6 & 8 \end{bmatrix}$$

Ad-Hoc cww_algorithm Example (10)

Algorithm

cww_merge_after_sort_recursive Function

Input:
numberZero: the number of 0s in the constant-weight word
onePositions: the positions in which to insert 1
Output:
result: the constant-weight word

```
1 function CWWMERGEAFTERSORTRECURSIVE(numberZero, onePositions)
2     zeroPositions ← [ ]
3     for i = 0 to numberZero do
4         zeroPositions[i] ← i
5     end for
6     onePositionSorted ← CWWSORTRECURSIVE(onePositions)
7     result ← CWWSORTMERGEBITS(zeroPositions, onePositionSorted)
8     return result
9 end function
```

- After *onePositionSorted* (line 6)

$$onePositionSorted = \begin{bmatrix} 0 & 4 & 6 & 8 \end{bmatrix}$$

- Before *result* (line 6)

$$zeroPositions = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \end{bmatrix}$$

$$onePositionSorted = \begin{bmatrix} 0 & 4 & 6 & 8 \end{bmatrix}$$

Call CWWSORTMERGEBITS

Ad-Hoc cww_algorithm Example (11)

Algorithm cww_sort_mergebits Function

Input:
zeroPositions: the positions of the 0s
onePositions: the positions in which to insert 1
Output:
result: the constant-weight word

```
1  function CWWSORT-MERGEBITS(zeroPositions, onePositions)
2      zeroQuadruples ← [ ]
3      for i = 0 to zeroPositions.size do
4          zeroQuadruples[i] ← ⟨zeroPositions[i], 0, 1, i⟩
5      end for
6      oneQuadruples ← [ ]
7      for j = 0 to onePositions.size do
8          oneQuadruples[j] ← ⟨onePositions[j] − j, 1, 0, j⟩
9      end for
10     quadruplesMerged ← MERGE(zeroQuadruples, oneQuadruples)
11     result ← [ ]
12     for i = 0 to quadruplesMerged.size do
13         ⟨_, bit, _, _⟩ ← quadruplesMerged[i]⟩]
14         result[i] ← bit
15     end for
16     return result
17 end function
```

- Input (line 1)

$$zeroPositions = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \end{bmatrix}$$

$$onePositions = \begin{bmatrix} 0 & 4 & 6 & 8 \end{bmatrix}$$

- Before *quadruplesMerged* (line 9)

$$zeroQuadruples = [\langle 0,0,1,0\rangle \ \langle 1,0,1,1\rangle \ \langle 2,0,1,2\rangle \ \langle 3,0,1,3\rangle \ \langle 4,0,1,4\rangle]$$

$$oneQuadruples = [\langle 0,1,0,0\rangle \ \langle 3,1,0,1\rangle \ \langle 4,1,0,2\rangle \ \langle 5,1,0,3\rangle]$$

- After *quadruplesMerged* (line 10)

$$quadruplesMerged = [\langle 0,1,0,0\rangle \ \langle 0,0,1,0\rangle \ \langle 1,0,1,1\rangle \ \langle 2,0,1,2\rangle \ \langle 3,1,0,1\rangle \ \langle 3,0,1,3\rangle \ \langle 4,1,0,2\rangle \ \langle 4,0,1,4\rangle \ \langle 5,1,0,3\rangle]$$

- After *result* (line 15)

$$result = [1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1]$$

Ad-Hoc cww_algorithm Example (12)

Algorithm

cww_merge_after_sort_recursive Function

Input:

numberZero: the number of 0s in the constant-weight word
onePositions: the positions in which to insert 1

Output:

result: the constant-weight word

```
1 function CWWMERGEAFTERSORTRECURSIVE(numberZero, onePositions)
2     zeroPositions ← [ ]
3     for i = 0 to numberZero do
4         zeroPositions[i] ← i
5     end for
6     onePositionSorted ← CWWSORTRECURSIVE(onePositions)
7     result ← CWWSORTMERGEBITS(zeroPositions, onePositionSorted)
8     return result
9 end function
```

- After *result* (line 7)

$$result = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Comparison Between the two Algorithm

The two algorithms have the same asymptotic complexity, $\mathcal{C}(n, m) = \mathcal{O}\Big((n+m) \cdot (\log{(n+m)})^2 + m \cdot (\log{(m)})^3\Big)$. However, if we evaluate the complexity at the constants, we see that, focusing only on the distinct parts of each algorithm's complexity, we have:

$$\begin{aligned} \textit{cww\_via\_insertionseries} \quad & 12 \cdot (n+m) + 21 \\ \textit{cww\_merge\_after\_sort\_recursive} \quad & 4 \cdot (n+m) + 10 \end{aligned}$$

This comparison makes it clear that the modified InsertionSeries approach offers better performance than the original, for the creation of a constant-weight word.

Using Parallelisation

The parallelisation technique can also be used in the construction of constant-weight words. However, we omit the details here, since the functions that benefit from parallelisation are the same as those used in the INSERTIONSERIES algorithm.

Constant-Time

This algorithm was also implemented using the constant-time approach.

All data-dependent branches were eliminated and replaced with mux bitmasks for conditional selection. All comparison and swap operations are branchless.
This approach allows us to minimise the possibility of side-channel attacks, as there are no data-dependent branches.