POLITECNICO
MILANO 1863

Constant Time, Parallel Shuffling

095947 - CRYPTOGRAPHY AND ARCHITECTURES FOR COMPUTER SECURITY

Conti Alessandro

Table of Contents

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

1. Introduction
    - Inserting Elements into an Array
2. Insertion Series
    - merge
        - Serial merge
        - Parallel merge
    - prefixsums
        - Serial prefixsums
        - Parallel prefixsums
    - insertionseries_sort_merge
    - insertionseries_sort_recursive
    - insertionseries_merge_after_sort_recursive
    - Insertion Series Conclusion
3. Constant-Weight Word
    - cww_via_insertionseries
    - Ad-Hoc cww_algorithm
        - cww_sort_mergebits
        - cww_sort_mergepos
        - cww_sort_recursive
        - cww_merge_after_sort_recursive
    - Constant-Weight Word Conclusion

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.

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

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.

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

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.

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

merge Function


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

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

merge Cost

Let $n :=$ `firstListSize` and $m :=$ `secondListSize`.

$$
\begin{aligned}
\mathcal{C}(n, m) &= \mathcal{C}_{malloc}(n+m) + \mathcal{C}_{memcpy}(n) + \mathcal{C}_{memcpy}(m) + \mathcal{C}_{qsort}(n+m) = \\
&= 1 + n + m + (n+m) \cdot \log(n+m) = \\
&= (n+m) + (n+m) \cdot \log(n+m) + 1 = \\
&= \mathcal{O}((n+m) \cdot \log(n+m))
\end{aligned}
$$

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

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

mergeParallel Function


```c
Quadruple *mergeParallel(Quadruple *firstList, size_t firstListSize, Quadruple *secondList, size_t secondListSize) {
    size_t resultSize = firstListSize + secondListSize;
    Quadruple *result = malloc(resultSize * sizeof(Quadruple));
    if (!result)
        return NULL;

#pragma omp parallel
    {
#pragma omp for schedule(static) nowait
        for (size_t i = 0; i < firstListSize; i++) {
            result[i + binarySearch(secondList, 0, secondListSize, firstList[i])] = firstList[i];
        }

#pragma omp for schedule(static)
        for (size_t i = 0; i < secondListSize; i++) {
            result[i + binarySearch(firstList, 0, firstListSize, secondList[i])] = secondList[i];
        }
    }

    return result;
}
```

mergeParallel Cost

Let n := `firstListSize`, m := `secondListSize` and t := number of threads used.

$$
\begin{aligned}
\mathcal{C}(n, m, t) &= \mathcal{C}_{malloc}(n+m) + \frac{n}{t} \cdot \mathcal{C}_{binarySearch}(m) + \frac{m}{t} \cdot \mathcal{C}_{binarySearch}(n) = \\
&= 1 + \frac{n}{t} \cdot \log(m) + \frac{m}{t} \cdot \log(n) = \\
&= \frac{n \cdot \log(m) + m \cdot \log(n)}{t} + 1 = \\
&= \mathcal{O}\left(\frac{n \cdot \log(m) + m \cdot \log(n)}{t}\right)
\end{aligned}
$$

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

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

prefixsums Function


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

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

prefixsums Cost

Let n := `list->listSize`.

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

$$
\begin{aligned}
\mathcal{S}(n) &= \mathcal{S}_{IntList}(n+1) + \mathcal{S}_{int} \ = \\
&= \ ((n+1) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + 4\,\mathrm{B} = \\
&= 4\,\mathrm{B} \cdot n + 32\,\mathrm{B} = \\
&= \mathcal{O}(n)
\end{aligned}
$$

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

prefixSumParallel Function


```c
IntList prefixSumParallel(const IntList *list) {
    IntList result;
    intlist_init(&result);
    intlist_reserve(&result, list->listSize + 1);

    size_t listSize = list->listSize;
    int *output = malloc((listSize + 1) * sizeof(int));
    output[0] = 0;

    int numberThreadUsed = 0;
    int *partialSumList = NULL;
#pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int numberThread = omp_get_num_threads();
#pragma omp single
        {
            numberThreadUsed = numberThread;
            partialSumList = calloc(numberThreadUsed, sizeof(int));
        }
        size_t chunk = (listSize + numberThread - 1) / numberThread;
        size_t start = threadID * chunk;
        size_t end = (start + chunk < listSize) ? start + chunk : listSize;

        int localSum = 0;
        for (size_t i = start; i < end; ++i) {
            localSum += list->list[i];
            output[i + 1] = localSum;
        }
        partialSumList[threadID] = localSum;
#pragma omp barrier
        int offset = 0;
        for (int i = 0; i < threadID; ++i)
            offset += partialSumList[i];
        for (size_t i = start + 1; i <= end; ++i)
            output[i] += offset;
    }
    for (size_t i = 0; i <= listSize; ++i)
        intlist_append(&result, output[i]);
    free(output);
    free(partialSumList);
    return result;
}
```

prefixSumParallel Cost

Let $n :=$ `list->listSize` and $t :=$ number of threads used.

$$\begin{aligned}
\mathcal{C}(n,t) &= \mathcal{C}_{intlist_init}(n+1) + \mathcal{C}_{intlist_reserve}(n+1) + \mathcal{C}_{malloc}(n+1) + \mathcal{C}_{omp_get_thread_num}(t) + \\
&\quad + \mathcal{C}_{omp_get_num_threads}(t) + \mathcal{C}_{calloc}(t) + 2 \cdot \frac{n}{t} + t + n + \mathcal{C}_{free}(n+1) + \mathcal{C}_{free}(t) = \\
&= 1 + \log(n+1) + 1 + 1 + 1 + 1 + 2 \cdot \frac{n}{t} + t + n + 1 + 1 = \\
&= 2 \cdot \frac{n}{t} + n + t + \log(n+1) + 7 = \\
&= \mathcal{O}(n)
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}(n,t) &= \mathcal{S}_{IntList}(n+1) + 3 \cdot \mathcal{S}_{size_t} + \mathcal{S}_{*int}(n+1) + 5 \cdot \mathcal{S}_{int} + \mathcal{S}_{*int}(t) = \\
&= ((n+1) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + 3 \cdot 8\,\mathrm{B} + (n+1) \cdot 4\,\mathrm{B} + 5 \cdot 4 + t \cdot 4\,\mathrm{B} = \\
&= 8\,\mathrm{B} \cdot n + 4\,\mathrm{B} \cdot t + 68\,\mathrm{B} = \\
&= \mathcal{O}(n+t)
\end{aligned}$$

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

insertionseries_sort_merge Function


```
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 Cost

Let $n :=$ `firstList->listSize` and $m :=$ `secondList->listSize`. For the parallel version, we have also $t :=$ number of threads used.

$$\begin{aligned}
\mathcal{C}(n,m) &= \mathcal{C}_{malloc}(n) + \mathcal{C}_{malloc}(m) + n + m + \mathcal{C}_{merge}(n,m) + \mathcal{C}_{intlist_init}(n+m) + \\
&\quad + \mathcal{C}_{intlist_reserve}(n+m) + (n+m)\cdot\mathcal{C}_{intlist_append}(n+m) + \\
&\quad + \mathcal{C}_{prefixSum}(n+m) + \mathcal{C}_{pairlist_init}(n+m) + \mathcal{C}_{pairlist_reserve}(n+m) + \\
&\quad + (n+m)\cdot\mathcal{C}_{pairlist_append}(n+m) + \mathcal{C}_{free}(n) + \mathcal{C}_{free}(m) + \mathcal{C}_{free}(n+m) + \\
&\quad + \mathcal{C}_{intlist_free}(n+m) + \mathcal{C}_{intlist_free}(n+m+1) = \\
&= 1 + 1 + n + m + \mathcal{C}_{merge}(n,m) + 1 + \log(n+m) + (n+m)\cdot\log(n+m+1) + \\
&\quad + \mathcal{C}_{prefixSum}(n+m) + 1 + \log(n+m) + (n+m)\cdot\log(n+m+1) + \\
&\quad + 1 + 1 + 1 + 1 + 1 = \\
&= 9 + n + m + 2\cdot\log(n+m) + (n+m)\cdot\log(n+m+1) + \\
&\quad + \mathcal{C}_{merge}(n,m) + \mathcal{C}_{prefixSum}(n+m)
\end{aligned}$$

This is the complexity of the function, but the function calls two functions that can be parallelised. On the next slide, we see the two complexities in the case of choosing either not to parallelise or to parallelise.

insertionseries_sort_merge Cost (2)

$$
\begin{aligned}
\mathcal{C}_{serial}(n, m) &= 9 + n + m + 2 \cdot \log(n+m) + (n+m) \cdot \log(n+m+1) + \\
&\quad + (n + m + (n+m) \cdot \log(n+m) + 1) + \\
&\quad + ((n+m+2) \cdot \log(n+m+1) + 1) = \\
&= 2n + 2m + (n+m+2) \cdot \log(n+m) + (2n+2m+2) \cdot \log(n+m+1) + 11 = \\
&= \mathcal{O}((n+m) \cdot \log(n+m))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_{parallel}(n, m, t) &= 9 + \frac{n}{t} + \frac{m}{t} + 2 \cdot \log(n+m) + (n+m) \cdot \log(n+m+1) + \\
&\quad + \left(\frac{n \cdot \log(m) + m \cdot \log(n)}{t} + 1\right) + \\
&\quad + \left(2 \cdot \frac{n}{t} + n + t + \log(n+1) + 7\right) = \\
&= (1 + {}^{3}\!/_{t}) \cdot n + {}^{m}\!/_{t} + t + \log(n+1) + \log\left(n^{{}^{m}\!/_{t}} \cdot m^{{}^{n}\!/_{t}}\right) + 2 \cdot \log(n+m) + \\
&\quad + (n+m) \cdot \log(n+m+1) + 17 = \\
&= \mathcal{O}((n+m) \cdot \log(n+m))
\end{aligned}
$$

insertionseries_sort_merge Cost (3)

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

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

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 Cost

Let $m :=$ `pairList->listSize`.

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

$$= \begin{cases} m & m \leq 1 \\ 2 \cdot (1 + \log(m/2)) + m \cdot \log(m/2 + 1) + 2 \cdot \mathcal{C}(m/2) + \\ \quad +\mathcal{C}_{insertionseries\_sort\_merge}(m/2, m/2) + 2 \cdot 1 & m > 1 \end{cases}$$

$$= \begin{cases} m & m \leq 1 \\ 2 \cdot \mathcal{C}(m/2) + 2 \cdot \log(m/2) + m \cdot \log(m/2 + 1) + \\ \quad +\mathcal{C}_{insertionseries\_sort\_merge}(m/2, m/2) + 4 & m > 1 \end{cases}$$

What is currently shown is the time complexity of an iteration; to find the total complexity, the master theorem must be applied. On the next slide, we see the complexity by considering only the big-O time complexity of the insertionseries_sort_merge function.

insertionseries_sort_recursive Cost (2)

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

insertionseries_sort_recursive Cost (3)

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

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

$$= \begin{cases} 8\,\mathrm{B} \cdot m + 16\,\mathrm{B} & m \leq 1 \\ 24\,\mathrm{B} \cdot m + 96\,\mathrm{B} & m > 1 \end{cases}$$

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

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

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

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

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

1. Introduction
   - Inserting Elements into an Array
2. Insertion Series
   - merge
     - Serial merge
     - Parallel merge
   - prefixsums
     - Serial prefixsums
     - Parallel prefixsums
   - insertionseries_sort_merge
   - insertionseries_sort_recursive
   - insertionseries_merge_after_sort_recursive
   - Insertion Series Conclusion
3. Constant-Weight Word
   - cww_via_insertionseries
   - Ad-Hoc cww_algorithm
     - cww_sort_mergebits
     - cww_sort_mergepos
     - cww_sort_recursive
     - cww_merge_after_sort_recursive
   - Constant-Weight Word Conclusion

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 Cost

Let n := `list->listSize` and m := `pairList->listSize`.

$$
\begin{aligned}
\mathcal{C}(n, m) &= \mathcal{C}_{pairlist\_init}(n) + n \cdot \mathcal{C}_{pairlist\_append}(n) + \mathcal{C}_{insertionseries\_sort\_recursive}(m) + \\
&\quad + \mathcal{C}_{insertionseries\_sort\_merge}(n, m) + \mathcal{C}_{intlist\_init}(n+m) + \mathcal{C}_{intlist\_reserve}(n+m) + \\
&\quad + m \cdot \mathcal{C}_{intlist\_append}(m) + \mathcal{C}_{pairlist\_free}(n) + \mathcal{C}_{pairlist\_free}(m) + \mathcal{C}_{pairlist\_free}(n+m) \\
&= 1 + n \cdot \log(n+1) + \mathcal{O}\left(m \cdot (\log(m))^2\right) + \mathcal{O}((n+m) \cdot \log(n+m)) + 1 + \\
&\quad + \log(n+m) + m \cdot \log(m+1) + 1 + 1 + 1 = \\
&= \mathcal{O}\left(m \cdot (\log(m))^2\right) + \mathcal{O}((n+m) \cdot \log(n+m)) + \\
&\quad + \log(n+m) + n \cdot \log(n+1) + m \cdot \log(m+1) + 5 = \\
&= \mathcal{O}\left((n+m) \cdot \log(n+m) + m \cdot (\log(m))^2\right)
\end{aligned}
$$

insertionseries_merge_after_sort_recursive Cost (2)

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

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

Comparison Between Naive and DJB Versions

Let $n :=$ size of the array into which new elements are to be inserted and $m :=$ list of elements to be inserted.

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

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

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 $\mathcal{O}(n \cdot \log(n))$ instead.

Using Parallelisation

In order to try to gain some computation time, it is possible to use the technique of parallelisation.
In this algorithm, the functions that can be parallelised are essentially the *merge* and the *prefixSum*. Parallelising these two functions resulted in an improvement:

- *prefixSum* went from having complexity $\mathcal{O}(n \cdot \log(n))$ to $\mathcal{O}(n)$, so a change from quasi-linear to linear complexity was achieved, and a very good asymptotic improvement was also obtained;
- *merge* function with the parallelisation has at least the same complexity as without the parallelisation, this similarity is due to the fact that they both have similar components, but for $t > 1$ there is a gain proportional to the number of threads. The two *merge* complexities are $\mathcal{O}((n+m) \cdot \log(n+m))$ and $\mathcal{O}\left(\frac{n \cdot \log(m) + m \cdot \log(n)}{t}\right)$.

1. Introduction
   - Inserting Elements into an Array
2. Insertion Series
   - merge
     - Serial merge
     - Parallel merge
   - prefixsums
     - Serial prefixsums
     - Parallel prefixsums
   - insertionseries_sort_merge
   - insertionseries_sort_recursive
   - insertionseries_merge_after_sort_recursive
   - Insertion Series Conclusion
3. Constant-Weight Word
   - cww_via_insertionseries
   - Ad-Hoc cww_algorithm
     - cww_sort_mergebits
     - cww_sort_mergepos
     - cww_sort_recursive
     - cww_merge_after_sort_recursive
   - Constant-Weight Word Conclusion

Application of Insertion Series

One application of the insertion series 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.

1. Introduction
   - Inserting Elements into an Array
2. Insertion Series
   - merge
     - Serial merge
     - Parallel merge
   - prefixsums
     - Serial prefixsums
     - Parallel prefixsums
   - insertionseries_sort_merge
   - insertionseries_sort_recursive
   - insertionseries_merge_after_sort_recursive
   - Insertion Series Conclusion
3. Constant-Weight Word
   - cww_via_insertionseries
   - Ad-Hoc cww_algorithm
     - cww_sort_mergebits
     - cww_sort_mergepos
     - cww_sort_recursive
     - cww_merge_after_sort_recursive
   - Constant-Weight Word Conclusion

cww_via_insertionseries Description

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

cww_via_insertionseries Function


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

cww_via_insertionseries Cost

Let $n :=$ `numberOfZero` and $m :=$ `positionOfOne->listSize`.

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

cww_via_insertionseries Cost (2)

$$
\begin{aligned}
\mathcal{S}(n, m) &= \mathcal{S}_{IntList}(n) + \mathcal{S}_{PairList}(m) + \mathcal{S}_{IntList}(n + m) = \\
&= (n \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + (m \cdot 8\,\mathrm{B} + 16\,\mathrm{B}) + ((n + m) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) = \\
&= 8\,\mathrm{B} \cdot n + 12\,\mathrm{B} \cdot m + 48\,\mathrm{B} = \\
&= \mathcal{O}(n + m)
\end{aligned}
$$

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

Ad-Hoc cww_algorithm

This algorithm represents an optimization of the previous one and, as such, also uses the MERGE and PREFIXSUMS 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.

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

cww_sort_mergebits Function


```
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_sort_mergebits Cost

Let $n :=$ `positionOfZero->listSize` and $m :=$ `positionOfOne->listSize`. For the parallel version, we have also $t :=$ number of threads used.

$$\begin{aligned}
\mathcal{C}(n, m) &= \mathcal{C}_{malloc}(n) + \mathcal{C}_{malloc}(m) + n + m + \mathcal{C}_{merge}(n, m) + \\
&+ \mathcal{C}_{intlist_init}(n + m) + \mathcal{C}_{intlist_reserve}(n + m) + \\
&+ (n + m) \cdot \mathcal{C}_{intlist_append}(n + m) + \\
&+ \mathcal{C}_{free}(n) + \mathcal{C}_{free}(m) + \mathcal{C}_{free}(n + m) = \\
&= 1 + 1 + n + m + \mathcal{C}_{merge}(n, m) + 1 + \log(n + m) + \\
&+ (n + m) \cdot \log(n + m + 1) + 1 + 1 + 1 = \\
&= 6 + n + m + \log(n + m) + (n + m) \cdot \log(n + m) + \mathcal{C}_{merge}(n, m)
\end{aligned}$$

This is the complexity of the function, but the function calls MERGE that can be parallelised. On the next slide, we see the two complexities in the case of choosing either not to parallelise or to parallelise.

cww_sort_mergebits Cost (2)

$$
\begin{aligned}
\mathcal{C}_{serial}(n, m) &= 6 + n + m + \log(n+m) + (n+m) \cdot \log(n+m) + \\
&\quad + ((n+m) + (n+m) \cdot \log(n+m) + 1) = \\
&= 2 \cdot (n+m) + \log(n+m) + 2 \cdot (n+m) \cdot \log(n+m) + 7 = \\
&= \mathcal{O}((n+m) \cdot \log(n+m))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_{parallel}(n, m, t) &= 6 + \frac{n}{t} + \frac{m}{t} + \log(n+m) + (n+m) \cdot \log(n+m) + \\
&\quad + \left(\frac{n \cdot \log(m) + m \cdot \log(n)}{t} + 1\right) = \\
&= \left(\frac{n}{t} + \frac{m}{t}\right) + \log(n+m) + \log\left(n^{m/t} \cdot m^{n/t}\right) + (n+m) \cdot \log(n+m) + 7 = \\
&= \mathcal{O}((n+m) \cdot \log(n+m))
\end{aligned}
$$

cww_sort_mergebits Cost (3)

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

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

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 Cost

Let $n :=$ `firstList->listSize` and $m :=$ `secondList->listSize`. or the parallel version, we have also $t :=$ number of thread used.

$$
\begin{aligned}
\mathcal{C}(n, m) = {} & \mathcal{C}_{malloc}(n) + \mathcal{C}_{malloc}(m) + n + m + \mathcal{C}_{merge}(n, m) + \\
& + 2 \cdot \mathcal{C}_{intlist\_init}(n + m) + 2 \cdot \mathcal{C}_{intlist\_reserve}(n + m) + \\
& + 2 \cdot (n + m) \cdot \mathcal{C}_{intlist\_append}(n + m) + \mathcal{C}_{prefixSum}(n + m) + \\
& + \mathcal{C}_{free}(n) + \mathcal{C}_{free}(m) + \mathcal{C}_{intlist\_free}(n + m) + \mathcal{C}_{intlist\_free}(n + m + 1) = \\
= {} & 1 + 1 + n + m + \mathcal{C}_{merge}(n, m) + 2 \cdot 1 + 2 \cdot \log(n + m) + \\
& + 2 \cdot (n + m) \cdot \log(n + m + 1) + \mathcal{C}_{prefixSum}(n + m) + 1 + 1 + 1 + 1 = \\
= {} & 8 + n + m + 2 \cdot \log(n + m) + 2 \cdot (n + m) \cdot \log(n + m + 1) + \\
& + \mathcal{C}_{merge}(n, m) + \mathcal{C}_{prefixSum}(n + m)
\end{aligned}
$$

This is the complexity of the function, but the function calls two functions that can be parallelised. On the next slide, we see the two complexities in the case of choosing either not to parallelise or to parallelise.

cww_sort_mergepos Cost (2)

$$\begin{aligned}
\mathcal{C}_{serial}(n,m) &= 8 + n + m + 2\cdot\log(n+m) + 2\cdot(n+m)\cdot\log(n+m+1) + \\
&\quad + (n + m + (n+m)\cdot\log(n+m) + 1) + ((n+2)\cdot\log(n+1)+1) = \\
&= 2\cdot(n+m) + 2\cdot\log(n+m) + (n+2)\cdot\log(n+1) + \\
&\quad + (n+m)\cdot\log(n+m) + 2\cdot(n+m)\cdot\log(n+m+1) + 10 = \\
&= \mathcal{O}((n+m)\cdot\log(n+m))
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}_{parallel}(n,m,t) &= 8 + \frac{n}{t} + \frac{m}{t} + 2\cdot\log(n+m) + 2\cdot(n+m)\cdot\log(n+m+1) + \\
&\quad + \left(\frac{n\cdot\log(m) + m\cdot\log(n)}{t} + 1\right) + \left(2\cdot\frac{n}{t} + n + t + \log(n+1) + 7\right) = \\
&= \left(\frac{3}{t}+1\right)\cdot n + \frac{m}{t} + t + \log(n+1) + \log\left(n^{m/t}\cdot m^{n/t}\right) + \\
&\quad + 2\cdot\log(n+m) + 2\cdot(n+m)\cdot\log(n+m+1) + 16 = \\
&= \mathcal{O}((n+m)\cdot\log(n+m))
\end{aligned}$$

cww_sort_mergepos Cost (3)

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

1. Introduction
   - Inserting Elements into an Array
2. Insertion Series
   - merge
     - Serial merge
     - Parallel merge
   - prefixsums
     - Serial prefixsums
     - Parallel prefixsums
   - insertionseries_sort_merge
   - insertionseries_sort_recursive
   - insertionseries_merge_after_sort_recursive
   - Insertion Series Conclusion
3. Constant-Weight Word
   - cww_via_insertionseries
   - Ad-Hoc cww_algorithm
     - cww_sort_mergebits
     - cww_sort_mergepos
     - cww_sort_recursive
     - cww_merge_after_sort_recursive
   - Constant-Weight Word Conclusion

cww_sort_recursive Function


```python
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 Cost

Let n := `intList->listSize`.

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

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

$$= \begin{cases} n & n \leq 1 \\ 2 \cdot \mathcal{C}\left(\frac{n}{2}\right) + 2 \cdot \log\left(\frac{n}{2}\right) + n \cdot \log\left(\frac{n}{2} + 1\right) + \mathcal{C}_{cww\_sort\_mergepos}\left(\frac{n}{2}, \frac{n}{2}\right) + 6 & n > 1 \end{cases}$$

What is currently shown is the time complexity of an iteration; to find the total complexity, the master theorem must be applied. On the next slide, we see the complexity by considering only the big-O time complexity of the cww_sort_mergepos function.

cww_sort_recursive Cost (2)

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

cww_sort_recursive Cost (3)

$$
\begin{aligned}
\mathcal{S}_{iteration}(n) &= \begin{cases} \mathcal{S}_{IntList}(n) & n \leq 1 \\ 2 \cdot \mathcal{S}_{size\_t} + 4 \cdot \mathcal{S}_{IntList}\left(\frac{n}{2}\right) + \mathcal{S}_{IntList}(n) & n > 1 \end{cases} \\
&= \begin{cases} 4\,\mathrm{B} \cdot n + 16\,\mathrm{B} & n \leq 1 \\ 2 \cdot 8\,\mathrm{B} + 4 \cdot \left(\frac{n}{2} \cdot 4\,\mathrm{B} + 16\,\mathrm{B}\right) + (n \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) & n > 1 \end{cases} \\
&= \begin{cases} 4\,\mathrm{B} \cdot n + 16\,\mathrm{B} & n \leq 1 \\ 12\,\mathrm{B} \cdot n + 96\,\mathrm{B} & n > 1 \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}(n) &= \sum_{i=0}^{\log n} \mathcal{S}_{iteration}\left(\frac{n}{2^i}\right) = \\
&= 12\,\mathrm{B} \cdot n \cdot \sum_{i=0}^{\log n} \frac{1}{2^i} + 96\,\mathrm{B} \cdot \sum_{i=0}^{\log n} 1 = \\
&< 12\,\mathrm{B} \cdot n \cdot \left(2 - \frac{1}{n}\right) + 96\,\mathrm{B} \cdot (\log n + 1) = \\
&= 24\,\mathrm{B} \cdot n + 96\,\mathrm{B} \cdot \log n + 84\,\mathrm{B} \\
&= \mathcal{O}(n)
\end{aligned}
$$

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

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 Cost

Let n := `numberOfZero` and m := `positionOfOne->listSize`.

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

cww_merge_after_sort_recursive Cost (2)

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

1 Introduction
- Inserting Elements into an Array

2 Insertion Series
- merge
  - Serial merge
  - Parallel merge
- prefixsums
  - Serial prefixsums
  - Parallel prefixsums
- insertionseries_sort_merge
- insertionseries_sort_recursive
- insertionseries_merge_after_sort_recursive
- Insertion Series Conclusion

3 Constant-Weight Word
- cww_via_insertionseries
- Ad-Hoc cww_algorithm
  - cww_sort_mergebits
  - cww_sort_mergepos
  - cww_sort_recursive
  - cww_merge_after_sort_recursive
- Constant-Weight Word Conclusion

Comparison Between the two Algorithm

The two algorithms have the same asymptotic complexity, $\mathcal{O}\Big((n+m)\cdot\log(n+m)+m\cdot(\log(m))^2\Big)$. 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:

$$\begin{array}{rl} \textit{cww\_via\_insertionseries} & \log(n\cdot m)+m\cdot\log(m+1)+2 \\ \textit{cww\_merge\_after\_sort\_recursive} & \log(n)+3 \end{array}$$

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.