Task 01:

After generating a random array of size `n` using the method `generate_array` which is filled with random integers from `0` to `2n` a block of parallel code execution using omp is started in line 73. However, the initial splitting is only supposed to be done exactly once which is why the term `#pragma omp single` is added to ensure not every thread executes this operation. \
The method `quicksort` receives an array as well as two indices and sorts the numbers in the given interval by their numeric value. The method `divide` performs the splitting of the array into two parts using a chosen pivot element and returns the new index of the pivot element which may now be at any index in the given interval. `quicksort` now recursively calls itself to also sort the new sublists above and below the returned position. To parallelize this `omp task` is used which creates a new subtask by the next code block with then is executed synchronously by the next available thread.

n	2 PEs	4 PEs	8 PEs
8	0.0000s	0.0001s	0.0000s
80	0.0000s	0.0002s	0.0000s
800	0.0003s	0.0006s	0.0006s
8000	0.0038s	0.0048s	0.0071s
80000	0.0312s	0.0436s	0.0735s
800000	0.4357s	0.7839s	1.4535s
8000000	4.1268s	6.3749s	12.2290s
80000000	172.1905s	130.3264s	92.3354s

Task 02:

To use radix sort to sort an array of integer numbers at each step of the iteration, the algorithm needs to know how many numbers with a `0` and how many numbers with a `1` at the current index are in the array. For that the method `one_amount` counts the latter and returns it from which the other amount can be calculated using the size of the array. This information can be used to now sort all numbers by putting them in the first or the second part of the list according to that information, while still keeping them in the same order according to all previous iterations. Using the number of elements in the two classes helps in finding the first indices and then just inserting the numbers into a new empty list and updating the indices as needed. The pointer of the new array later is copied to the array to be sorted. \
To parallelize each thread, works on a small subpart of the array to count the class amounts and later to insert them into the new array. A prefix sum has to be used to find the starting indices for each class in each thread.

n	2 PEs	4 PEs	8 PEs
8	0.0015s	0.0019s	0.0032s
80	0.0017s	0.0022s	0.0033s
800	0.0013s	0.0016s	0.0032s
8000	0.0028s	0.0025s	0.0041s
80000	0.0136s	0.0087s	0.0084s

n	2 PEs	4 PEs	8 PEs
800000	0.1426s	0.0964s	0.0949s
8000000	1.5822s	1.1046s	0.8035s
80000000	15.7717s	10.9164s	7.9710s

Task 04:

```
def find_min_pancake_num(initial_config):
    n = size of initial_config
    # general upper bound for pancake problem (proof omitted)
    best = ceil(18 / 11 * n)

    # stack to save a configuration and the amount of steps it took to get there
    s: Stack of pairs (pancake configuration plus an integer number)

    push the pair (initial_configuration, 0) onto s

    while s not empty:
        current_config, num_steps = pop top element from s
        if num_steps > best:
            # stop current iteration if the current iteration can not beat best found solution
            continue
        else:
            parallel for i from 2 to n:
                next_config = flip top i pancakes of current_config
                if next_config is sorted and num_steps + 1 < best:
                    atomic update best = num_steps + 1
                else:
                    push the pair (next_config, num_steps + 1) onto s
    return best
```

Exercise 1 (Unimplemented parts)

We didn't implement the Parallel Partition successfully because of some technical details. But I will give the pseudocode of the parallel partition below. It may look verbose and complicated, but the key point is to find the correct index to insert the elements. **(See line 32. - 34.)**

Parallel Partition

```
 1: procedure ParallelPartition($A$, $n$, pivot, offset)
 2:     ▷ The following 3 arrays are used for storing the count of element in each processors, that are
        smaller/equal/larger to pivot.
 3:     localSmaller ← Array[1..p]
 4:     localEqual ← Array[1..p]
 5:     localBigger ← Array[1..p]
 6:     ▷ $B$ as a temporary array to store the rearranged array
 7:     $B$ ← Array[1..n]
 8:     for tid=1..p in parallel do
 9:         smallerCount ← 0
10:         equal ← 0
11:         biggerCount ← 0
12:         for j=start(tid)...end(tid) do
13:             if $A[j + \text{offset}] < \text{pivot}$ then
14:                 smallerCount$++$
15:                 else if $A[j + \text{offset}] == \text{pivot}$ then
16:                     equal$++$
17:                 end
18:                 else
19:                     biggerCount$++$
20:                 end
21:             end
22:         end
23:         localSmaller[tid] ← smallerCount
24:         localEqual[tid] ← equal
25:         localBigger[tid] ← biggerCount
26:     end
27:     prefixSmaller ← pPrefAdd(localSmaller)
28:     prefixEqual ← pPrefAdd(localEqual)
29:     prefixBigger ← pPrefAdd(localBigger)
30:     ▷ Let prefix*[0] be 0
31:     for tid=1..p in parallel do
32:         sIdx ← prefixSmaller[tid $-$ 1]
33:         eIdx ← prefixSmaller[p] $+$ prefixEqual[tid $-$ 1]
34:         bIdx ← prefixSmaller[p] $+$ prefixEqual[p] $+$ prefixBigger[tid $-$ 1]
35:         for j=start(tid)...end(tid) do
36:             if $A[j + \text{offset}] < \text{pivot}$ then
37:                 $B[\text{sIdx}++] \leftarrow A[j + \text{offset}]$
38:                 else if $A[j + \text{offset}] == \text{pivot}$ then
39:                     $B[\text{eIdx}++] \leftarrow A[j + \text{offset}]$
40:                 end
41:                 else
42:                     $B[\text{bIdx}++] \leftarrow A[j + \text{offset}]$
43:                 end
44:             end
45:         end
46:     end
47:     Copy $B$ back to $A[\text{offset}..\text{offset} + n]$
48: end
```

Exercise 3

3.1

The **insert** operation can result in concurrent write if it is not carefully designed, given the case where $\mathbf{index}(A[i]) == \mathbf{index}(A[j])$.

The other parts are guaranteed to be thread-safe.

3.2

For a parallel version of **copy**(A, B), we will need the help of another array C.

```
procedure PARALLEL_COPY($A$, $B$)
    ▷ $C$ stores the size of each bucket in $B$
    $C \leftarrow$ Array[1..nb]
    pfill$(C, 0)$
    for i=1..nb in parallel do
        $C[i] \leftarrow \mathbf{len}(B[i])$
    end
    $P \leftarrow \mathbf{pPrefAdd}(C)$
    for i=1..nb-1 in parallel do
        ▷ we set $P[0]$ as 0
        beg $\leftarrow P[i-1]+1$
        end $\leftarrow P[i]$
        for j=beg..end in parallel do
            $A[j] \leftarrow B[i][j - \text{beg}]$
        end
    end
end
```

3.2.1 Comments

- For the beg and end in the above algorithm, we note that $(P[i] - (P[i-1]+1)+1)$ is equal to $C[i]$, namely $\mathbf{len}(B[i])$.
- I don't see the necessity of using **pfill**(). It's likely used for avoiding data corruption.

3.2.2 Complexity Analysis

Operation	*Time Complexity*	*Processors Required*
Get the lengths	$O(1)$	nb
Parallel Prefix Sum	$O(\log \text{nb})$	nb/2
Insert	$O(1)$	n
Overall	$O(\log \text{nb})$	n

For fast Parallel Bucket Sort, we need enough number of buckets. However, when it comes to the parallel copy operation, a lower nb is preferred. Herein lies a tradeoff.

Exercise 5

(a)

Given an index i, to find the final position j of a_i in the merged list, we can use Binary Search to find the biggest element b_k in B that is smaller than a_i (or the smallest element in B that is bigger than a_i), which requires $O(\log n)$ comparisons. (See section **Parallel Merge sort** in the lecture slide for example) Then, $j = i + k$, as there are $i + k$ elements in total that are smaller than a_i.

(b)

Denote by $\mathbf{computeInsertPos}(A, B, i)$ the algorithm described in (a), which returns the final position j to insert our $a_i = A[i]$.

```
procedure PARALLEL_MERGE($A$,$B$)
    $C \leftarrow$ Array$[1..2n]$
    for $i = 1..n$ in parallel do
        ▷ $j_1 \neq j_2$ as the elements are unique
        $j_1 \leftarrow \mathbf{computeInsertPos}(A, B, i)$
        $j_2 \leftarrow \mathbf{computeInsertPos}(B, A, i)$
        $C[j_1] \leftarrow A[i]$
        $C[j_2] \leftarrow B[i]$
    end
end
```

(b.1) Comments

- If the elements are not unique and $j_1 == j_2$ by accident, we can set j_2 as $j_1 + 1$ to avoid data race.
- The total work T_1 is apparently $O(n \log n)$ and the span T_∞ is $O(\log n)$.

(c)

This task seems impossible from the software perspective. With $O(n)$ processors, we can compare a_i with $b_{1..n}$ in $O(1)$. (but don't we need to broadcast a_i to all the processors?)

That said, it's still not possible to reduce the results in $O(1)$. Consider the following 2 cases:

1. $\max\limits_{\{k : a_i \geq b_k\}} k$;
2. $\sum\limits_{\kappa \in \{k : a_i \geq b_k\}} 1$.

In both cases, the reduction takes $O(\log \#\{k : a_i \geq b_k\})$. $O(1)$ is only made possible if we have some hardware "hacks" or some magic.

If such a hack exists, we can use $n \cdot O(n) = O(n^2)$ processors for merging two lists in $O(1)$.