

Weekly Challenge 04: Divide-and-Conquer vs Traditional Sorting

CS/CE 412/471 Algorithms: Design and Analysis

Spring 2025

Objective

In this WC, you will:

- design a divide-and-conquer (D&C) version of the **Bubble Sort** algorithm,
- implement both the D&C and traditional versions of Bubble Sort, and
- compare their runtimes for different input sizes and analyze their performance.

Motivation

Any comparison based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case. Parallel computing can further reduce the running time of certain algorithms. In its most basic sense, parallel computing refers to the simultaneous execution of tasks using multiple processors and cores.

Algorithms can be parallelized when some of their steps can be computed in parallel. Lack of dependencies implies potential for parallel execution. For example, divide-and-conquer algorithms are often well-suited for parallelization. The division of the problem into *subproblems that can be solved independently* facilitates parallel execution.

Learning to implement an algorithm for parallel execution in a programming language of your choice is beyond the scope of this WC. However, we will design a divide-and-conquer algorithm that *could be* parallelized. And we will make do with a conventional, serial implementation of it.

Task

- Implement the bubble sort algorithm (you may use a third party implementation).
- Design a D&C version of bubble sort that could be parallelized, i.e., its subproblems can be solved independently of each other. See the notes below for further instructions.
- Implement (for serial execution) your D&C algorithm.
- Measure and plot the run time of both algorithms on randomly generated inputs of increasing size: 100, 200, 300, 400, \dots , N , where N is determined by the performance of your machine and your D&C algorithm.

D&C Bubble Sort

This is not a standard algorithm and there is no right or wrong answer, provided your algorithm:

- correctly sorts the input,
- follows the D&C paradigm:
 - divides the problem into subproblems,
 - recursively conquers the subproblems, and
 - optionally, combines the sub-solutions into the final solution.
- uses the bubble sort algorithm in a meaningful manner, and
- is suitable for parallelization, i.e. as much as possible, involves solving independent problems.

Submission

Submit as separate files:

1. your code file containing documented implementations of your algorithms,
2. your plot comparing the run time of the two algorithms on the same inputs, and
3. a report containing an analysis of your algorithm and the timings results:
 - a description of your algorithm, especially how it fulfills the requirements listed in the previous section,
 - which approach is faster and why, and
 - how the D&C approach could be further optimized.

Weekly Challenge 04 Report: Performance Analysis of Bubble Sort and Divide-and-Conquer Bubble Sort

bq08283

CS/CE 412/471 Algorithms: Design Analysis
Habib University
Spring 2025

1 Algorithm Description

1.1 Divide-and-Conquer Bubble Sort

The D&C Bubble Sort improves efficiency by dividing the array into smaller subarrays, sorting them recursively (left_half then right_half) and then merging these sorted array. Below is a step-by-step breakdown of the implementation:

```
def dnc_bubble_sort(arr):  
    if len(arr) <= 10: # Base case: apply Bubble Sort  
        return bubble_sort(arr)  
  
    mid = len(arr) // 2  
  
    #recursively sorts both halves  
    left_half = dnc_bubble_sort(arr[:mid])  
    right_half = dnc_bubble_sort(arr[mid:])  
  
    #merges the sorted halves  
    return merge_sorted_arrays(left_half, right_half)
```

Figure 1: D&C Bubble Sort Algorithm

This algorithm fulfills the Divide-and-Conquer requirements as follows:

- **Divide:** The array is split into two independent subarrays

```
mid = len(arr) //2
```

- **Conquer:** Each subarray is recursively sorted using the same D&C approach

```
left_half = dnc.bubble_sort(arr[:mid])
```

```
right_half = dnc.bubble_sort(arr[mid:])
```

- **Merge:** The sorted subarrays are merged efficiently

```
return merge_sorted_arrays(left_half, right_half)
```

```
def merge_sorted_arrays(left, right):
    merged = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    merged.extend(left[i:])
    merged.extend(right[j:])

    return merged
```

Figure 2: D&C Bubble Sort Merge Algorithm

- **Uses Bubble Sort Algorithm:** By calling the function

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr
```

Figure 3: D&C Bubble Sort Algorithm

- **Parallelization:** This algorithm can be parallelised to optimise speed because the subarrays are independent.

2 Performance Comparison

The runtime comparison graph (Figure 4) illustrates the execution time of both sorting algorithms for different input sizes.

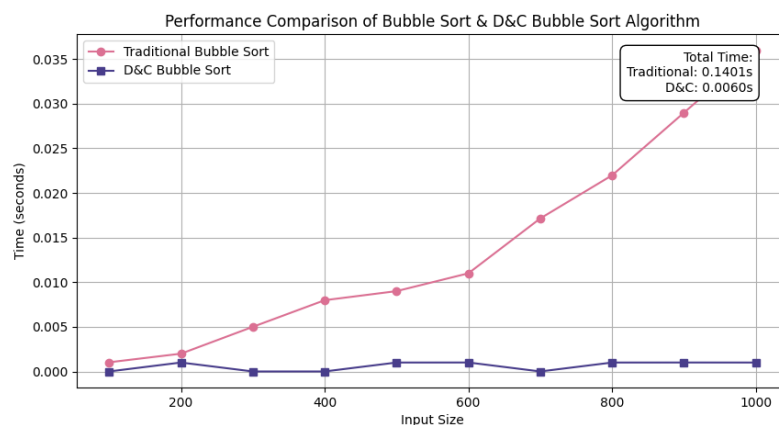


Figure 4: Performance Comparison of Bubble Sort and D&C Bubble Sort

3 Analysis of Results

3.1 Graph Trends

The graph demonstrates that the **D&C Bubble Sort** maintains a flat, constant runtime across all input sizes, demonstrating its efficiency and scalability, whereas the **Traditional Bubble Sort**'s runtime rises sharply with input size, creating a distinct upward curve.

3.2 Complexity Analysis

Traditional Bubble Sort

- **Best Case:** $O(n)$: Stops early after one pass; already sorted out.
- **Worst Case:** $O(n^2)$: Reverse-sorted, requiring maximal swaps and $n - 1$ passes.
- **Average Case:** $O(n^2)$: Randomly shuffled, multiple passes and swaps.

Divide & Conquer (D&C) Bubble Sort

- **Best Case:** $O(n \log n)$: It takes less time to merge inputs that have been sorted.
- **Worst Case:** $O(n^2 \log n)$: Deep recursion and $O(n^2)$ merging were used to reverse-sort.
- **Average Case:** $O(n^2 \log n)$: Recursive splits and costly merges dominate.

3.3 Which Approach is Faster and Why?

The **D&C Bubble Sort** is quicker because of its effective divide-and-conquer method and reduced complexity. In the best case scenario, the **D&C Bubble Sort** completes in just 0.006 seconds with a relatively flat curve $O(n \log n)$ whereas the **Traditional Bubble Sort** takes 0.140 seconds to complete and increases sharply with input size $O(n^2)$. For bigger inputs, the D&C technique is much more scalable since it eliminates duplicate operations by using Bubble Sort only once per sub-array and merging effectively.

3.4 Optimizing the D&C Bubble Sort Approach

While the D&C Bubble Sort is more efficient, further optimizations could enhance its performance:

- **Parallel Processing:** The recursive sorting steps can be parallelised by using multi-threading or multiprocessing.

- **Hybrid Sorting:** Utilising a hybrid strategy in which Insertion Sort is used to sort smaller subarrays for increased efficiency.
- **In-Place Merging:** Optimising memory usage by minimising the amount of auxiliary space utilised during merging.

4 References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.
2. <https://www.geeksforgeeks.org/bubble-sort-algorithm/>
3. <https://www.geeksforgeeks.org/merge-two-sorted-arrays/>
4. <https://builtin.com/articles/timing-functions-python>

```

import time
import random
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import interp1d
from scipy.signal import savgol_filter

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr

def merge_sorted_arrays(left, right):
    merged = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    merged.extend(left[i:])
    merged.extend(right[j:])

    return merged

def dnc_bubble_sort(arr):
    if len(arr) <= 10: # Base case: apply Bubble Sort
        return bubble_sort(arr)

    mid = len(arr) // 2

    #recursively sorts both halves
    left_half = dnc_bubble_sort(arr[:mid])
    right_half = dnc_bubble_sort(arr[mid:])

    #merges the sorted halves
    return merge_sorted_arrays(left_half, right_half)

# implementation of measuring time
def measure_time(sort_function, arr):
    start_time = time.time()
    sort_function(arr.copy())
    return time.time() - start_time

sizes = list(range(100, 1100, 100))
bubble_sort_times = []
dnc_bubble_sort_times = []

```

```

for size in sizes:
    test_array = [random.randint(0, 10000) for _ in range(size)]

    bubble_sort_time = measure_time(bubble_sort, test_array)
    dnc_bubble_sort_time = measure_time(dnc_bubble_sort, test_array)

    bubble_sort_times.append(bubble_sort_time)
    dnc_bubble_sort_times.append(dnc_bubble_sort_time)

total_bubble_sort_time = sum(bubble_sort_times)
total_dnc_bubble_sort_time = sum(dnc_bubble_sort_times)

# # Smooth Curve using Savitzky-Golay Filter
# sizes_np = np.array(sizes)
# bubble_times_np = np.array(bubble_times)
# dc_bubble_times_np = np.array(dc_bubble_times)

# window_length = min(11, len(sizes_np) - 1 if len(sizes_np) % 2 == 0 else
len(sizes_np))

# smooth_bubble_times = savgol_filter(bubble_times_np, window_length=window_length,
polyorder=3)
# smooth_dc_bubble_times = savgol_filter(dc_bubble_times_np,
window_length=window_length, polyorder=3)

# Plot the Results
plt.figure(figsize=(10, 5))
plt.plot(sizes, bubble_sort_times, label="Traditional Bubble Sort",
marker='o',color='palevioletred')
plt.plot(sizes, dnc_bubble_sort_times, label="D&C Bubble Sort",
marker='s',color='darkslateblue')
plt.xlabel("Input Size")
plt.ylabel("Time (seconds)")
plt.title("Performance Comparison of Bubble Sort & D&C Bubble Sort Algorithm")
plt.text(
    0.95, 0.95,
    f"Total Time:\nTraditional: {total_bubble_sort_time:.4f}s\nD&C:
{total_dnc_bubble_sort_time:.4f}s",
    horizontalalignment='right',
    verticalalignment='top',
    transform=plt.gca().transAxes,
    fontsize=10,
    bbox=dict(facecolor='white', edgecolor='black', boxstyle='round,pad=0.5')
)
plt.legend()
plt.grid()
plt.savefig('WC04_bq08283.png')
plt.show()

"""
References
https://www.geeksforgeeks.org/bubble-sort-algorithm/
https://www.geeksforgeeks.org/bubble-sort-algorithm/
https://www.geeksforgeeks.org/merge-two-sorted-arrays/
https://builtin.com/articles/timing-functions-python
https://www.geeksforgeeks.org/merge-two-sorted-arrays/
"""

```