VIET NAM NATIONAL UNIVERSITY

HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Seminar Report

Subject: Sorting Algorithms

Course: CS163 - Data Structures

Group 05 - 24A02:

Thuan, Le Minh - 24125105

Phu, Nguyen Minh - 24125103

Lac, Quach Thien - 24125092

Quan, Pham Nguyen Minh - 24125041

Supervisors:

MSc. Thanh, Ho Tuan

MSc. Loc, Truong Phuoc

April 1, 2025

Contents

List of Tables

List of Figures

Listings

1 Heap Sort

1.1 Introduction

History: Heap Sort was first introduced by J.W.J. Williams in 1964. This innovation aimed to exploit the heap data structure to improve sorting performance.

Figure 1: J. W. J. Williams (1930-2012)

Definition: Heap Sort is a comparison-based, in-place sorting algorithm that leverages the properties of a heap data structure—specifically, a max heap when sorting in ascending order. Its basic premise is:

1. **Max Heap Property:** In a max heap, every parent node is greater than or equal to its children. This guarantees that the largest element is always at the root.
2. **In-Place Sorting:** Heap Sort rearranges the data within the input array, meaning it does not require significant additional memory.
3. **Time Complexity:** It consistently operates in $O(n \log n)$ time in the worst case, making it reliable and predictable.

1.2 Algorithm and Implementation

Step 1: Build the Max Heap

- **Determine the Starting Point:**
  - The array is treated as a complete binary tree.

- The last non-leaf node resides at index $\lfloor n/2 \rfloor - 1$, where n is the total number of elements.

- **Call heapify() on Each Non-Leaf Node:**
  - Starting from the last non-leaf node and moving upward to the root, apply the heapify() procedure to enforce the max heap property.
  - The heapify() procedure ensures that for a node at index i, the value at i is greater than or equal to its children.

Algorithm 1 Heapify

```
Input: Array A, index i, size n
Output: Array A with max heap property enforced at index i
function HEAPIFY(A, i, n)
    largest ← i
    left ← 2i + 1
    right ← 2i + 2
    if left < n and A[left] > A[largest] then
        largest ← left
    end if
    if right < n and A[right] > A[largest] then
        largest ← right
    end if
    if largest ≠ i then
        SWAP(A[i], A[largest])
        HEAPIFY(A, largest, n)
    end if
end function
```

Step 2: Sort the Array (Extract Elements from Heap)

- **Swap the Root with the Last Element:**
  - The root of the max heap (index 0) holds the maximum value. Swap it with the last element in the heap (at index $n - 1$).

- **Reduce Heap Size:**
  - After the swap, consider the last element as sorted. Reduce the effective heap size by one so that it is excluded from further heap operations.

- **Re-heapify the Root:**

  - Call the heapify() procedure on the root (index 0) to restore the max heap property over the reduced heap.

- **Repeat Extraction:**

  - Continue this process, swapping the root with the last unsorted element and re-heapifying until the entire array is sorted.

Algorithm 2 HeapSort

Input: Array A
Output: Sorted array A
function HEAPSORT(A)
    $n \leftarrow |A|$
    for $i \leftarrow \lfloor n/2 \rfloor - 1$ **downto** 0 **do** ▷ Build Max Heap
        HEAPIFY(A, i, n)
    end for
    for $i \leftarrow n - 1$ **downto** 1 **do**
        SWAP($A[0]$, $A[i]$)
        HEAPIFY(A, 0, i)
    end for
end function

Below is the implementation in C++:

```cpp
void heapify(std::vector<int>& arr, int i, int n) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        heapify(arr, largest, n);
    }
}

```

```cpp
void heapSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; --i)
        heapify(arr, i, n);
    for (int i = n - 1; i > 0; --i) {
        std::swap(arr[0], arr[i]);
        heapify(arr, 0, i);
    }
}
```

Listing 1: Heap Sort in C++

Below is the implementation in Python:

```python
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)
```

Listing 2: Heap Sort in Python

1.3 Evaluation

1.3.1 Building the Heap

Operation: Constructing a Heap (Max-Heap) from the input array.

Complexity: $O(n)$

Explanation: While a single "heapify" operation takes $O(\log n)$, the total time to build the heap is $O(n)$ due to optimization when handling elements at lower levels of the heap.

1.3.2 Extracting Elements from the Heap

Operation: Extract the root element (maximum) and rearrange the heap.

Complexity: $O(n \log n)$

Explanation: There are n extractions, and each extraction is followed by a "heapify" operation that takes $O(\log n)$.

1.3.3 In-Place Sorting

Operation: Swap the extracted root element with the last unsorted element.

Complexity: $O(n \log n)$

Explanation: Swapping is combined with extraction and re-heapification, contributing to the overall complexity.

1.3.4 Space Complexity

Space Used: $O(1)$

Explanation: Heap Sort is an in-place sorting algorithm, so it does not require additional space beyond the input array.

1.4 Application

Heap sort is a strong algorithm. It has several practical applications due to its efficiency and predictable performance:

1. **Sorting Large Datasets:** Heap sort is particularly useful for sorting large datasets where memory usage is a concern. *Example:* Sorting a large database of employee records by salary.

2. **Priority Queue Implementation:** Heap sort is the foundation for implementing priority queues. Priority queues are used in scheduling algorithms, such as in operating systems or task management systems. *Example:* Scheduling tasks in a CPU based on their priority levels.

3. **External Sorting:** Heap sort is used in external sorting algorithms where data is too large to fit into memory. It helps in merging sorted chunks of data efficiently. *Example:* Sorting large files stored on disk.

4. **Real-Time Systems:** Heap sort is used in real-time systems where predictable performance is critical. *Example:* Sorting sensor data in real-time for autonomous vehicles.

5. **Selection Algorithms:** Heap sort is used in selection algorithms to find the kth smallest or largest element in a list. *Example:* Finding the median of a dataset efficiently.

6. **Game Development:** Heap sort is used in game development for managing and sorting game objects based on priority or distance. *Example:* Rendering objects in a game based on their distance from the camera.

7. **Event-Driven Simulations:** Heap sort is used in simulations where events need to be processed in a specific order. *Example:* Simulating the behavior of a queue in a bank or a call center.

8. **Operating Systems:** Heap sort is used in operating systems for memory management and process scheduling. *Example:* Managing memory allocation for processes in a multi-tasking environment.

1.5 Problems

1.5.1 Kth Largest Element in an Array

LeetCode

Description: Given an unsorted array, determine the kth largest element.

Detailed Instructions:

1. Create a min-heap (priority queue) and insert the first k elements of the array into it.

2. For every remaining element in the array, compare it with the heap's root. If it's larger, remove the root and insert the new element, maintaining the heap property.

3. Once all elements are processed, the root of the heap is the kth largest element. Use this technique to optimize your time complexity to $O(n \log k)$.

1.5.2 Find Median from Data Stream

LeetCode

Description: Continuously find the median of a stream of numbers.

Detailed Instructions:

1. Use two heaps: a max-heap for the lower half of the numbers and a min-heap for the upper half.

2. When a new number arrives, decide which heap to insert it into, then rebalance the heaps if their sizes differ by more than one.

3. For median retrieval, if heaps are equal in size, average the roots; otherwise, return the root of the larger heap. This dual-heap method ensures that the median is found efficiently after each insertion.

1.5.3 Merge k Sorted Lists

LeetCode

Description: Merge multiple sorted linked lists into one sorted linked list.

Detailed Instructions:

1. Initialize a min-heap and insert the head (first node) of each of the k lists into it.

2. Repeatedly extract the smallest node from the heap and append it to the merged list. If the extracted node has a next node, insert that node into the heap.

3. Continue until the heap is empty. This process ensures that at each step, you're adding the smallest current node, maintaining overall sorted order.

1.5.4 Sliding Window Median

LeetCode

Description: Given an array and a window size k, compute the median of every sliding window.

Detailed Instructions:

1. Use two heaps (max-heap and min-heap) to represent the current window's lower and upper halves.

2. As the window slides, add the new element and remove the element that's moving out of the window. Rebalance the heaps to maintain the required size properties.

3. After every slide, compute the median based on the roots of the two heaps. This ensures an efficient update and retrieval of the median as the window moves.

2 Odd-Even Transposition Sort

2.1 Introduction

History: Developed in 1972 by N. Habermann for parallel processor arrays, OETS gained prominence through Kenneth Batcher's work on sorting networks. Its design enables efficient execution on architectures with local interconnections, predating modern GPU-based sorting techniques.

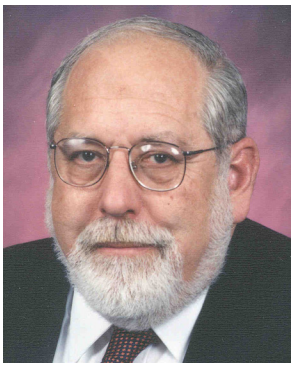

Figure 2: Kenneth Batcher (1935-2019)

Definition: Odd-Even Transposition Sort (OETS) is a parallel sorting algorithm derived from Bubble Sort, using alternating comparison phases to enable efficient concurrent execution. It operates in $O(n^2)$ time sequentially but achieves $O(n)$ time with n processors in parallel implementations. The algorithm's simplicity and parallelizability make it valuable for multi-core systems and educational demonstrations.

2.2 Algorithm and Implementation

- **Array:** The algorithm operates on a 1D array, comparing and swapping adjacent elements in-place.
- **Parallel Processors:** Each processor holds one element and communicates with immediate neighbors during odd/even phases.

Algorithm 3 Odd-Even Transposition Sort

Input: Array A of size n
Output: Sorted array A
function OddEvenSort(A, n)
    $isSorted \leftarrow$ False
    while $\neg isSorted$ **do**
        $isSorted \leftarrow$ True
        for $i \leftarrow 1$ to $n-1$ **step** 2 **do**
            if $A[i] > A[i+1]$ **then**
                swap($A[i], A[i+1]$)
                $isSorted \leftarrow$ False
            end if
        end for
        for $i \leftarrow 0$ to $n-1$ **step** 2 **do**
            if $A[i] > A[i+1]$ **then**
                swap($A[i], A[i+1]$)
                $isSorted \leftarrow$ False
            end if
        end for
    end while
end function

Below is the implementation in C++:

```cpp
void oddEvenSort(int arr[], int n) {
    bool isSorted = false;
    while (!isSorted) {
        isSorted = true;
        for (int i = 1; i < n - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                std::swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
        for (int i = 0; i < n - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                std::swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
    }
}
```

Listing 3: Odd-Even Transposition Sort in C++

Below is the implementation in Python:

```python
def odd_even_sort(arr):
    n = len(arr)
    is_sorted = False
    while not is_sorted:
        is_sorted = True
        for i in range(1, n - 1, 2):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                is_sorted = False
        for i in range(0, n - 1, 2):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                is_sorted = False
    return arr
```

Listing 4: Odd-Even Transposition Sort in Python

2.3 Evaluation

2.3.1 Building the Even-Odd Transposition Sort Array

Operation: Create and set up the array for sorting the input elements.

Complexity: $O(n)$

Explanation: The initial setup involves creating an array of size n (where n is the number of elements to be sorted). This is a straightforward operation that requires linear time to allocate memory for the array.

2.3.2 Filling the Even-Odd Transposition Sort Array

Operation: Distribute elements from the input array into the sorting process.

Complexity: $O(n)$

Explanation: The algorithm traverses through the input array to perform comparisons and swaps. Each element is compared with its adjacent elements in both the even and odd phases. This requires a linear traversal of the array, leading to a time complexity of $O(n)$ for each phase.

2.3.3 Even Phase of the Sort

Operation: Compare and swap adjacent elements at even indices.

Complexity: $O(n)$

Explanation: In the even phase, the algorithm iterates through the array, comparing pairs of elements at even indices. Each comparison and potential swap takes constant time, resulting in a linear time complexity for this phase.

2.3.4 Odd Phase of the Sort

Operation: Compare and swap adjacent elements at odd indices.

Complexity: $O(n)$

Explanation: Similar to the even phase, the odd phase involves iterating through the array and comparing pairs of elements at odd indices. This also results in a linear time complexity.

2.3.5 Overall Sorting Process

Operation: Repeat the even and odd phases until the array is sorted.

Complexity: $O(n^2)$ (in the average and worst cases)

Explanation: The even and odd phases are repeated until no swaps are made. In the worst case, the algorithm may need to perform n passes through the array, leading to a quadratic time complexity of $O(n^2)$.

2.3.6 Space Complexity

Operation: Memory usage during sorting.

Complexity: $O(1)$

Explanation: The Even-Odd Transposition Sort is an in-place sorting algorithm, meaning it does not require additional storage proportional to the input size. Only a constant amount of extra space is used for variables (like loop counters and flags).

2.4 Applications

While the even-odd transposition sort is not commonly used in practical applications due to its inefficiency compared to other sorting algorithms, it has some niche applications, particularly in

educational contexts and parallel computing:

- **Educational Purposes:** The algorithm is often used in computer science courses to teach sorting algorithms and the concept of parallel processing. It helps students understand how sorting can be performed in a distributed manner.
- **Parallel Computing:** In environments where multiple processors are available, the even-odd transposition sort can be implemented to take advantage of parallelism. Each phase (even and odd) can be executed simultaneously on different processors, making it suitable for parallel architectures.
- **Sorting Networks:** The even-odd transposition sort is related to sorting networks, which are used in hardware implementations of sorting algorithms. These networks can be used in applications where sorting needs to be done quickly and efficiently in hardware.

2.5 Problems

2.5.1 Sorting: Bubble Sort

HackerRank

Description: Given an array of integers, perform a bubble sort and count the number of swaps that are made. Print the number of swaps and the first and last elements of the sorted array.

Detailed Instructions:

1. Initialize Variables: Create a variable to count swaps and initialize it to zero.
2. Use a nested loop: the outer loop runs from 0 to $n-1$, and the inner loop runs from 0 to $n-i-1$.
3. Compare adjacent elements and swap them if they are in the wrong order.
4. Increment the swap counter each time a swap is made.
5. Output the Results: After sorting, print the total number of swaps, the first element, and the last element of the sorted array.
6. Return the Output.

2.5.2 A. Sort the Array

Codeforces

Description: You are given an array of integers. You need to determine if it is possible to sort the array by performing a certain number of operations. Each operation consists of choosing two indices and swapping the elements at those indices.

Detailed Instructions:

1. Check for Sortedness: If the array is already sorted, output "YES".
2. Count Odd and Even Elements: If the array is not sorted, check if the odd and even indexed elements can be sorted independently.
3. Sort the Array: Create a copy of the array and sort it.
4. Compare: Check if the sorted array can be obtained by sorting the original array's odd and even indexed elements separately.
5. Output the Result: If it is possible to sort the array, print "YES"; otherwise, print "NO".

2.5.3 Merge Intervals

LeetCode

Description: Given a collection of intervals, merge all overlapping intervals.

Detailed Instructions:

1. Sort the Intervals: First, sort the intervals based on the starting times.
2. Initialize a Result List: Create an empty list to hold the merged intervals.
3. Iterate Through the Sorted Intervals:
   - For each interval, check if it overlaps with the last interval in the result list.
   - If it does, merge them by updating the end of the last interval in the result list.
   - If it does not, add the current interval to the result list.
4. Return the Merged Intervals: After processing all intervals, return the result list.

3 Counting Sort

3.1 Introduction

History: Counting Sort was developed by Harold H. Seward in the early 1950s, often cited around 1954. It is recognized for its ability to sort integers in linear time under the right conditions.

Figure 3: Harold H. Seward (1930-2012)

Definition: Counting Sort is a non-comparison-based sorting algorithm that sorts elements by counting the number of occurrences of each unique value in the input array. It then uses these counts to determine the positions of each element in the final, sorted array.

- The algorithm operates in linear time.
- Time complexity: $O(n+k)$, where n is the number of elements and k is the range of the input values.

3.2 Algorithm and Implementation

1. **Determine the Range of the Data:** Sometimes min is assumed to be 0 to simplify the process.
2. **Initialize the Count Array:** Create a count array of size $(\max - \min + 1)$ and initialize all elements to 0.
3. **Count the Occurrences:** Iterate over the input array and for each element, increment the corresponding index in the count array. For an element x, increment count$[x - \min]$.

4. **Transform the Count Array into a Cumulative Count Array:** Modify the count array such that each element at index i contains the sum of previous counts. This cumulative count tells you the final position of each element in the output array.

5. **Build the Output Array:**

   - Traverse the input array once more, usually in reverse order to ensure stability (maintaining the original order of equal elements).
   - Place each element x into its correct position in the output array by using the cumulative count, then decrement the count value for x.

Algorithm 4 Counting Sort

```
Input: Array A of size n
Output: Sorted array B
function CountingSort(A, n)
    min ← min(A)
    max ← max(A)
    count ← array of size (max − min + 1) initialized to 0
    for each element x in A do
        count[x − min] ← count[x − min] + 1
    end for
    for i ← 1 to length(count) − 1 do
        count[i] ← count[i] + count[i − 1]
    end for
    B ← array of size n
    for i ← 0 to n − 1 do
        x ← A[i]
        B[count[x − min] − 1] ← x
        count[x − min] ← count[x − min] − 1
    end for
    Return B
end function
```

Below is the implementation in C++:

```cpp
std::vector<int> countingSort(const std::vector<int>& arr) {
    int n = arr.size();
    int MAX = arr[0], MIN = arr[0];

    for (int i = 0; i < n; ++i) {
        MAX = std::max(MAX, arr[i]);
        MIN = std::min(MIN, arr[i]);
    }

    int range = MAX - MIN + 1;
    std::vector<int> count(range, 0);

    for (int i = 0; i < n; ++i)
        ++count[arr[i] - MIN];
    for (int i = 1; i < range; ++i)
        count[i] += count[i - 1];

    std::vector<int> sorted(n);

    for (int i = n - 1; i >= 0; --i) {
        --count[arr[i] - MIN];
        sorted[count[arr[i] - MIN]] = arr[i];
    }

    return sorted;
}
```

Listing 5: Counting Sort in C++

Below is the implementation in Python:

```python
def counting_sort(arr):
    n = len(arr)
    if n == 0:
        return []

    MAX = max(arr)
```

```
    MIN = min(arr)

    range_of_elements = MAX - MIN + 1
    count = [0] * range_of_elements

    for i in range(n):
        count[arr[i] - MIN] += 1

    for i in range(1, range_of_elements):
        count[i] += count[i - 1]

    sorted_arr = [0] * n
    for i in range(n - 1, -1, -1):
        sorted_arr[count[arr[i] - MIN] - 1] = arr[i]
        count[arr[i] - MIN] -= 1

    return sorted_arr
```

Listing 6: Counting Sort in Python

3.3 Evaluation

3.3.1 Building the Count Array

Operation: Create and populate the count array with the frequency of each element in the input array.

Complexity: $O(n)$

Explanation: Each element in the input array is processed once to record its frequency in the count array.

3.3.2 Modifying the Count Array

Operation: Convert the count array into a cumulative count array, indicating the position of each element in the sorted order.

Complexity: $O(\max - \min + 1)$

Explanation: For each value in the count array, we compute a running total (cumulative sum), which takes linear time with respect to the range of values.

3.3.3 Reconstructing the Sorted Array

Operation: Use the count array to place each element from the input array into its correct position in the output array, preserving their order (stability).

Complexity: $O(n)$

Explanation: Each element from the input array is placed in the output array based on its position derived from the cumulative count.

3.3.4 Space Complexity Evaluation

Space Used: $O(\max - \min + 1)$

Explanation: Additional space is needed to store the count array. Its size depends on the range of values in the input array $(\max - \min + 1)$.

3.4 Application

Counting sort is a non-comparison-based sorting algorithm that works well for sorting integers or objects with small, discrete key ranges. It counts the occurrences of each element and uses this information to place elements in their correct sorted position.

1. **Sorting Small Range Integers:** Counting sort is highly efficient for sorting integers or keys within a small, known range. *Example:* Sorting exam scores (e.g., 0 to 100) or ages of individuals.

2. **Histogram Generation:** Counting sort can be used to generate histograms or frequency distributions of data. *Example:* Analyzing the frequency of words in a text or the distribution of pixel intensities in an image.

3. **Data Compression:** Counting sort is used in data compression algorithms to analyze and organize data frequencies. *Example:* Building frequency tables for Huffman coding or other compression techniques.

4. **Counting Occurrences:** Counting sort can be used to count the occurrences of elements in a dataset. *Example:* Counting the number of students who scored a particular grade in an exam.

5. **Sorting Characters or Small Alphabets:** Counting sort is efficient for sorting characters or small alphabets. *Example:* Sorting letters in a word or DNA sequences (A, T, C, G).

6. **Real-Time Systems:** Counting sort is used in real-time systems where sorting needs to be done quickly and efficiently. *Example:* Sorting sensor data in real-time for IoT devices.

7. **Database Indexing:** Counting sort can be used in database systems to sort and index records with small key ranges. *Example:* Sorting records by a small set of categories or flags.

8. **Statistical Analysis:** Counting sort is used in statistical analysis to sort and analyze data distributions. *Example:* Sorting survey responses or experimental data for further analysis.

9. **String Sorting:** Counting sort is used in string sorting algorithms, especially when sorting by a specific character position. *Example:* Sorting strings lexicographically or by a specific attribute.

3.5 Problems

3.5.1 Sort Colors

LeetCode

Description: Given an array with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

Detailed Instructions:

1. Initialize Pointers: Set three pointers: low, mid, and high. Initialize low and mid to the start of the array and high to the end.

2. Iterate Through the Array:

   - While mid is less than or equal to high:
   - If array[mid] is 0, swap it with array[low], increment both low and mid.

- If array[mid] is 1, just increment mid.
- If array[mid] is 2, swap it with array[high] and decrement high.

3. Complete the Sorting: Continue until mid exceeds high.
4. Return the Sorted Array.

3.5.2 Counting Sort 1

HackerRank

Description: Count the frequency of each integer in an array.

Detailed Instructions:

1. Identify the maximum value in the array to determine the size of your count array.
2. Traverse the input array, incrementing the corresponding index in the count array for each element encountered.
3. Output the count array, ensuring that the frequency for every number (including those with zero occurrences) is reported.

3.5.3 The Full Counting Sort

HackerRank

Description: Sort pairs of numbers and strings based on numeric keys while preserving the original order for equal keys.

Detailed Instructions:

1. Create a count array based on the numeric keys and count the occurrences.
2. Convert the count array into a cumulative frequency array to determine the final positions of the keys.
3. Place each (number, string) pair into a new array at the position determined by the cumulative counts, ensuring that elements with the same key retain their original order (stability).

3.5.4 Counting Sort 2

HackerRank

Description: Efficiently sort an array of integers using counting sort.

Detailed Instructions:

1. Find the maximum integer in the input array to allocate a count array of size $\max + 1$.
2. Iterate through the array to populate the count array with frequencies of each integer.
3. Reconstruct the sorted array by iterating through the count array and appending each integer based on its count, ensuring linear time performance when the range isn't excessively large.

3.5.5 Find All Numbers Disappeared in an Array

LeetCode

Description: Identify which numbers in the range [1, n] are missing from the input array.

Detailed Instructions:

1. Create a boolean (or count) array of size $n + 1$, initialized to mark all numbers as missing.
2. Traverse the input array and mark each number that appears in the corresponding index of your boolean array.
3. Iterate through the range 1 to n and collect those numbers that remain unmarked, as these are the missing numbers. This counting technique ensures that every potential number is checked exactly once.

4 Pigeonhole Sort

4.1 Introduction

History: Peter Gustav Lejeune Dirichlet, who formally stated the idea in 1834.

Figure 4: Peter Gustav Lejeune Dirichlet (1805-1859)

Definition:

- Pigeonhole sorting is a sorting algorithm that is suitable for sorting lists of elements where the number of elements and the number of possible key values are approximately the same.
- It requires $O(n + \text{Range})$ time where n is the number of elements in the input array and 'Range' is the number of possible values in the array.

4.2 Algorithm and Implementation

Pigeonhole sort is similar to counting sort, but differs in that it "moves items twice: once to the bucket array and again to the final destination."

1. **Determine the range:** Find the minimum and maximum values in the array. Let the minimum and maximum values be 'min' and 'max' respectively. Also, find the range as 'max-min+1'.
2. **Initialize an array:** Set up an array of initially empty "pigeonholes" the same size as the range.

3. **Distribution step:** Visit each element of the array and then put each element in its pigeonhole. An element $arr[i]$ is put in the hole at index $arr[i] - \min$.

4. **Output reconstruction phase:** Start the loop all over the pigeonhole array in order and put the elements from non-empty holes back into the original array.

Algorithm 5 Pigeonhole Sort

```
1: Input: Array A of size n
2: Output: Sorted array A
3: function PigeonholeSort(A, n)
4:     min ← min(A)
5:     max ← max(A)
6:     range ← max − min + 1
7:     P ← array of size range initialized to 0
8:     for each element x in A do
9:         P[x − min] ← P[x − min] + 1
10:    end for
11:    index ← 0
12:    for i ← 0 to range − 1 do
13:        while P[i] > 0 do
14:            A[index] ← i + min
15:            index ← index + 1
16:            P[i] ← P[i] − 1
17:        end while
18:    end for
19: end function
```

Below is the implementation in C++:

```cpp
void pigeonholeSort(int arr[], int n) {
    int MIN = arr[0], MAX = arr[0];

    for (int i = 1; i < n; i++) {
        MIN = std::min(MIN, arr[i]);
        MAX = std::max(MAX, arr[i]);
    }

    int range = MAX - MIN + 1;
    std::vector<int> holes(range, 0);

    for (int i = 0; i < n; i++)
        ++holes[arr[i] - MIN];

    int index = 0;
    for (int i = 0; i < range; i++) {
        while (holes[i]--) {
            arr[index] = i + MIN;
            ++index;
        }
    }
}
```

Listing 7: Pigeonhole Sort in C++

Below is the implementation in Python:

```python
def pigeonhole_sort(arr):
    MIN = min(arr)
    MAX = max(arr)
    range_size = MAX - MIN + 1

    holes = [0] * range_size

    for num in arr:
        holes[num - MIN] += 1

```

```
    index = 0
    for i in range(range_size):
        while holes[i] > 0:
            arr[index] = i + MIN
            index += 1
            holes[i] -= 1
```

Listing 8: Pigeonhole Sort in Python

4.3 Evaluation

4.3.1 Building the Pigeonhole Array

Operation: Create and set up pigeonholes (buckets) for sorting the input array.

Complexity: $O(n + \text{range})$

Explanation: Each element in the input array is placed into its respective pigeonhole. The range refers to the difference between the maximum and minimum values.

4.3.2 Filling the Pigeonhole Array

Operation: Distribute elements from the input array into the pigeonholes.

Complexity: $O(n)$

Explanation: Traverse through the input array and assign each element to the appropriate pigeonhole.

4.3.3 Reconstructing the Sorted Array

Operation: Rebuild the sorted array by traversing through the pigeonholes in order.

Complexity: $O(n + \text{range})$

Explanation: Iterate over the pigeonholes and transfer elements back to the input array, ensuring the order is correct.

4.3.4 Space Complexity Evaluation

Space Used: $O(\text{range})$

Explanation: Additional memory is required to store the pigeonhole array, with its size being determined by the range of values in the input array (maximum value - minimum value + 1).

4.4 Application

Pigeonhole sort is a non-comparison-based sorting algorithm that works well for sorting integers or objects with small, discrete key ranges. It counts the occurrences of each element and uses this information to place elements in their correct sorted position.

1. **Sorting Small Range Integers:** Pigeonhole sort is highly efficient for sorting integers or keys within a small, known range. *Example:* Sorting exam scores (e.g., 0 to 100) or ages of individuals.

2. **Histogram Generation:** Pigeonhole sort can be used to generate histograms or frequency distributions of data. *Example:* Analyzing the frequency of words in a text or the distribution of pixel intensities in an image.

3. **Data Compression:** Pigeonhole sort is used in data compression algorithms to analyze and organize data frequencies. *Example:* Building frequency tables for Huffman coding or other compression techniques.

4. **Counting Occurrences:** Pigeonhole sort can be used to count the occurrences of elements in a dataset. *Example:* Counting the number of students who scored a particular grade in an exam.

5. **Sorting Characters or Small Alphabets:** Pigeonhole sort is efficient for sorting characters or small alphabets. *Example:* Sorting letters in a word or DNA sequences (A, T, C, G).

6. **Real-Time Systems:** Pigeonhole sort is used in real-time systems where sorting needs to be done quickly and efficiently. *Example:* Sorting sensor data in real-time for IoT devices.

7. **Database Indexing:** Pigeonhole sort can be used in database systems to sort and index records with small key ranges. *Example:* Sorting records by a small set of categories or flags.

8. **Statistical Analysis:** Pigeonhole sort is used in statistical analysis to sort and analyze data distributions. *Example:* Sorting survey responses or experimental data for further analysis.

9. **String Sorting:** Pigeonhole sort is used in string sorting algorithms, especially when sorting by a specific character position. *Example:* Sorting strings lexicographically or by a specific attribute.

4.5 Problems

4.5.1 Pigeonhole Sort 1

Description: Sort an array of integers using pigeonhole sort.

Detailed Instructions:

1. Identify the minimum and maximum values in the array to determine the range.

2. Initialize a pigeonhole array of size $\max - \min + 1$.

3. Traverse the input array, placing each element in the corresponding pigeonhole.

4. Reconstruct the sorted array by iterating through the pigeonhole array and placing elements back into the original array.

4.5.2 The Full Pigeonhole Sort

Description: Sort pairs of numbers and strings based on numeric keys while preserving the original order for equal keys.

Detailed Instructions:

1. Create a pigeonhole array based on the numeric keys and place the elements in the corresponding pigeonholes.

2. Reconstruct the sorted array by iterating through the pigeonhole array and placing elements back into the original array, ensuring stability.

4.5.3 Find All Numbers Disappeared in an Array

Description: Identify which numbers in the range [1, n] are missing from the input array.

Detailed Instructions:

1. Create a boolean (or pigeonhole) array of size $n+1$, initialized to mark all numbers as missing.
2. Traverse the input array and mark each number that appears in the corresponding index of your boolean array.
3. Iterate through the range 1 to n and collect those numbers that remain unmarked, as these are the missing numbers. This counting technique ensures that every potential number is checked exactly once.

4.5.4 Sort an Array of 0s, 1s and 2s

geeksforgeeks.org

Description: Sort an array containing only 0s, 1s, and 2s (a variant often used for the Dutch National Flag problem).

Detailed Instructions:

1. Iterate over the array to count the number of 0s, 1s, and 2s.
2. Overwrite the original array by placing the counted number of 0s first, followed by 1s and then 2s.
3. Validate the resulting array to ensure that all elements are in the expected sorted order.

4.5.5 Sort Characters By Frequency

LeetCode

Description: Rearrange the characters of a string so that characters with higher frequencies come first.

Detailed Instructions:

1. Traverse the string to build a frequency map for each character.

2. Use a bucket sort (pigeonhole sort style) where each bucket corresponds to a frequency, placing characters into the appropriate bucket.

3. Reconstruct the string by iterating through the buckets from highest to lowest frequency, appending each character as many times as its frequency.

4.5.6 Sort Array by Increasing Frequency

LeetCode

Description: Sort the elements of an array by their frequency. In case of ties, the smaller number should come first.

Detailed Instructions:

1. Count the frequency of each element using a hash map or count array.

2. Create buckets where each bucket holds the elements corresponding to a specific frequency.

3. Iterate through the buckets in ascending order of frequency; for tied frequencies, sort the numbers in ascending order, then rebuild the array from these sorted buckets.

4.5.7 Sort Integers by The Number of 1 Bits

LeetCode

Description: Sort integers by the number of 1s in their binary representation; if two numbers have the same number of 1s, sort them by their numeric value.

Detailed Instructions:

1. For each integer, compute the number of 1 bits (this can be done using bit manipulation or a built-in function).

2. Use these counts as keys for sorting. If two numbers have the same bit count, compare their numerical values.

3. Reconstruct the sorted array based on these comparisons, ensuring that the algorithm is efficient by possibly caching bit counts for repeated values.

5 Comparison of Sorting Algorithms

Algorithm	**Stability**	**Time Complexity**	**Space Complexity**	**Pros**	**Cons**
Heap Sort	Not stable	$O(n \log n)$	$O(1)$	Stable performance, memory efficient	Not stable, complex implementation
Counting Sort	Stable	$O(n + k)$	$O(n + k)$	Linear time, simple, stable	Memory intensive, not flexible for wide value ranges
Pigeonhole Sort	Potentially stable	$O(n + k)$	$O(n + k)$	Efficient for small value ranges, simple, fast	Memory intensive, reduced efficiency for wide value ranges, less common in practice
Odd-Even Transition Sort	Stable	$O(n^2)$	$O(1)$	Simple, easy to implement, efficient for small arrays	Poor performance on single-core processors or large arrays

Table 1: Comparison of Sorting Algorithms

6 Quizzes

1. **What is Heapify in Heap Sort?**

   (a) Building a complete binary tree

   (b) Ensuring the max heap or min heap property

   (c) Sorting the array by comparing each element

   (d) Creating an AVL tree

   Answer: (b) Ensuring the max heap or min heap property

   Explanation:

   - Heapify is the process of ensuring that a binary tree satisfies the max heap property (parent is greater than or equal to its children) or the min heap property (parent is less than or equal to its children).

   - In Heap Sort, Heapify is used to build and maintain the heap structure during the sorting process.

   - For example, if a parent node has a smaller value than its child, Heapify will swap their values to ensure the heap property is maintained.

2. **Which of the following is not an application of Heap Sort?**

   (a) Sorting large datasets

   (b) Implementing priority queues

   (c) Sorting data in external memory

   (d) Sorting data with a small range of values

   Answer: (d) Sorting data with a small range of values

   Explanation:

   - Heap Sort is a comparison-based sorting algorithm with a time complexity of O(n log n) in all cases.

   - It is suitable for sorting large datasets (a), implementing priority queues (b), and sorting data in external memory (c).

- However, Heap Sort is not ideal for sorting data with a small range of values (d) because algorithms like Counting Sort or Pigeonhole Sort are more efficient in such cases.

3. **When does Counting Sort work most efficiently?**

   (a) When the range of values (k) is larger than the number of elements (n)

   (b) When the range of values (k) is smaller than or approximately equal to the number of elements (n)

   (c) When the data consists of real numbers

   (d) When the data consists of strings

   Answer: (b) When the range of values (k) is smaller than or approximately equal to the number of elements (n)

   Explanation:

   - Counting Sort is a non-comparison-based sorting algorithm with a time complexity of O(n + k), where k is the range of values in the data.
   - It works most efficiently when k is smaller than or approximately equal to n (the number of elements).
   - If k is too large, Counting Sort becomes inefficient because it requires additional memory to store the count array.

4. **Which of the following is not an application of Counting Sort?**

   (a) Sorting integers within a small range

   (b) Generating histograms from data

   (c) Sorting large datasets with a large range of values

   (d) Sorting characters in a string

   Answer: (c) Sorting large datasets with a large range of values

   Explanation:

   - Counting Sort is suitable for sorting integers within a small range (a), generating histograms (b), and sorting characters in a string (d).
   - However, it is not suitable for sorting large datasets with a large range of values (c) because its space complexity is O(k), and if k is too large, it will consume excessive memory.

5. **When does Pigeonhole Sort work most efficiently?**

   (a) When the number of elements (n) is larger than the range of values (k)

   (b) When the number of elements (n) is approximately equal to the range of values (k)

   (c) When the data consists of real numbers

   (d) When the data consists of strings

   Answer: (b) When the number of elements (n) is approximately equal to the range of values (k)

   Explanation:

   - Pigeonhole Sort is a non-comparison-based sorting algorithm with a time complexity of O(n + k), where k is the range of values in the data.
   - It works most efficiently when the number of elements (n) is approximately equal to the range of values (k).
   - If k is too large, Pigeonhole Sort becomes inefficient because it requires additional memory to store the "pigeonholes."

6. **How is Pigeonhole Sort different from Counting Sort?**

   (a) Pigeonhole Sort moves elements twice

   (b) Counting Sort moves elements twice

   (c) Pigeonhole Sort does not require a range of values

   (d) Counting Sort does not require a range of values

   Answer: (a) Pigeonhole Sort moves elements twice

   Explanation:

   - Both Pigeonhole Sort and Counting Sort are non-comparison-based sorting algorithms with a time complexity of O(n + k).
   - However, Pigeonhole Sort differs in that it moves elements twice:
   + First, elements are placed into their corresponding "pigeonholes."
   + Second, elements are retrieved from the "pigeonholes" and placed back into the original array in sorted order.
   - In contrast, Counting Sort moves elements only once when building the output array.

7. **Which algorithm is most suitable for sorting integers within a small range?**

   (a) Heap Sort

   (b) Counting Sort

   (c) Pigeonhole Sort

   (d) Both (b) and (c)

   Answer: (d) Both (b) and (c)

   Explanation:

   - Both Counting Sort and Pigeonhole Sort are non-comparison-based sorting algorithms with a time complexity of O(n + k), where k is the range of values in the data.
   - They work most efficiently when k is smaller than or approximately equal to n (the number of elements).
   - Heap Sort (a) is a comparison-based algorithm with a time complexity of O(n log n), making it less suitable for small ranges compared to Counting Sort and Pigeonhole Sort.

8. **What is the average time complexity of Odd-Even Transition Sort?**

   (a) O(n)

   (b) O(n log n)

   (c) $O(n^2)$

   (d) O(log n)

   Answer: (c) $O(n^2)$

   Explanation:

   - The average time complexity of Odd-Even Transition Sort is $O(n^2)$, similar to other simple sorting algorithms like Bubble Sort.

9. **How does the comparison and swapping process work in Odd-Even Transition Sort?**

   (a) Compare and swap elements at even positions with each other, then elements at odd positions with each other.

(b) Compare and swap elements at even positions with elements at odd positions.

(c) Compare and swap adjacent elements.

(d) Compare and swap elements at a fixed distance apart.

Answer: (b) Compare and swap elements at even positions with elements at odd positions

Explanation:

- In Odd-Even Transition Sort: - During the even phase, pairs like (0, 1), (2, 3), (4, 5), etc., are compared and swapped if necessary. - During the odd phase, pairs like (1, 2), (3, 4), (5, 6), etc., are compared and swapped if necessary.

10. **Can Odd-Even Transition Sort be parallelized?**

(a) Yes, because comparisons and swaps in even and odd phases can be performed simultaneously.

(b) No, because comparisons and swaps must be performed sequentially.

(c) Only on specialized hardware.

(d) Only when the array is nearly sorted.

Answer: (a) Yes, because comparisons and swaps in even and odd phases can be performed simultaneously

Explanation:

- Odd-Even Transition Sort can be parallelized because the comparisons and swaps in the even and odd phases can be executed concurrently on different processors.

11. **Which category of sorting algorithms does Odd-Even Transition Sort belong to?**

(a) Comparison-based sorting.

(b) Non-comparison-based sorting.

(c) Stable sorting.

(d) Unstable sorting.

Answer: (a) Comparison-based sorting

Explanation:

- Odd-Even Transition Sort is a comparison-based sorting algorithm because it relies on comparing elements to determine their order.

A Appendix

- Source code: GitHub
- Template source: khongsomeo
- Reference: tranlynhathao
- Documentation: The algorithms bundle