Analysis of Algorithms

BLG 335E

Project 2 Report

Yusuf Yıldız
yildizyu21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 14.12.2023

1. Implementation

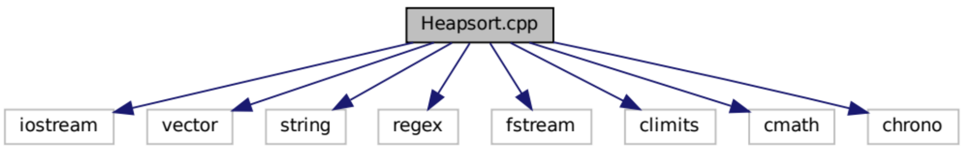
1.1. General Outline of the Implementation

[IMPORTANT]
Usage: ./Heapsort [DatasetFileName].csv [FunctionName] [OutputFileName].csv [AdditionalParameters - i, d, k]
File names must not contain the path of the files, just give the file name. Code is written to read the data from, and write output to './Data/' folder.

This file contains the implementation of the max heap building and sorting algorithms.

```
#include <iostream>
#include <vector>
#include <string>
#include <regex>
#include <fstream>
#include <climits>
#include <cmath>
#include <chrono>
```

Include dependency graph for Heapsort.cpp:



Classes

- struct Population

    Represents a population data structure for sorting.

Functions

- void quickSwap (std::vector< Population > &vec, int i1, int i2)

    Swaps two elements in a vector.
- bool validate_arguments (int argc, char **argv)

    Validates command line arguments.
- void removeBOM (std::ifstream &file)

Removes the BOM(Byte Order Mark) from the input file.

- void write_to_csv (const std::string fileName, const std::vector< Population > &vec, char mode, const std↩ ::string out)

  Writes the vector to a csv file.
- void read_from_csv (const std::string fileName, std::vector< Population > &vec)

  Reads the csv file and creates a vector of Population.
- void max_heapify (std::vector< Population > &vec, int i, int size, const std::string fileName, bool writeToFile)

  Creates the binary max heap structure for given index recursively.
- void build_max_heap (std::vector< Population > &vec, int size, const std::string fileName, bool writeToFile)

  Builds a binary max heap from the vector.
- void heapsort (std::vector< Population > &vec, int size, const std::string fileName)

  Sorts the vector using heapsort algorithm.
- void max_heap_insert (std::vector< Population > &vec, std::string city, int key, const std::string fileName)

  Inserts a new element to the heap.
- Population heap_extract_max (std::vector< Population > &vec, const std::string fileName)

  Extracts the maximum element from the heap.
- void heap_increase_key (std::vector< Population > &vec, int i, int key, const std::string fileName, bool write↩ ToFile)

  Increases the key of the element at given index.
- Population heap_maximum (std::vector< Population > &vec, const std::string fileName)

  Returns the maximum element of the heap.
- void dary_max_heapify (std::vector< Population > &vec, int i, int size, int d, const std::string fileName, bool writeToFile)

  Creates the d-ary max heap structure for given index recursively.
- void dary_build_max_heap (std::vector< Population > &vec, int size, int d, const std::string fileName, bool writeToFile)

  Builds a d-ary max heap from the vector.
- int dary_calculate_height (int size, int d, const std::string fileName)

  Calculates the height of the d-ary heap.
- Population dary_extract_max (std::vector< Population > &vec, int d, const std::string fileName)

  Extracts the maximum element from the d-ary heap.
- void dary_insert_element (std::vector< Population > &vec, std::string city, int key, int d, const std::string fileName)

  Inserts a new element to the d-ary heap.
- void dary_increase_key (std::vector< Population > &vec, int i, int key, int d, const std::string fileName, bool writeToFile)

  Increases the key of the element at given index.
- int main (int argc, char **argv)

  The main function that orchestrates the max heap process based on command line arguments.
- void dary_heapsort (std::vector< Population > &vec, int size, int d, const std::string fileName)

  Sorts the vector using heapsort algorithm.

1.2. Analysis of the Implemented Functions

In this section, we will further analyze the implementation details of the functions and also analyze the complexities of the functions.

max_heapify()

```
void max_heapify (
            std::vector< Population > & vec,
            int i,
            int size,
            const std::string fileName,
            bool writeToFile )
```

Creates the binary max heap structure for given index recursively.

Parameters

vec	Vector to be built as a max heap.
i	Index of the element to be max heapified.
size	Size of the vector.
fileName	Name of the file to be written.
writeToFile	Whether to write the output to file or not.

Time Complexity Analysis:

The `max_heapify` function is part of the heap sort algorithm, and its primary purpose is to maintain the max-heap property. The time complexity of `max_heapify` is primarily determined by the height of the heap, and it is logarithmic with respect to the number of elements in the heap (n). The worst-case time complexity is $O(\log n)$, where n is the number of elements.

- **Worst-case:** The worst-case scenario occurs when the element at index i violates the max-heap property, and it needs to be swapped down to the leaf level of the heap. This process takes $O(\log n)$ time, as the height of a binary heap is $\log n$.
- **Best-case:** The best-case scenario happens when the subtree rooted at node i already satisfies the max-heap property. In this case, the function completes in constant time, $O(1)$.

Real-World Usage Example:

In a real-world scenario, the `max_heapify` function can be employed to efficiently find and extract the element with the maximum value from a dataset. For example, it could be utilized to maintain a priority queue of cities based on their populations. This proves useful in applications such as:

- Identifying the most populated cities.
- Managing priority tasks in a scheduling system.
- Finding the top-performing items in an e-commerce platform.

The efficiency of `max_heapify` is particularly advantageous when dealing with large datasets where finding the maximum element quickly is essential.

build_max_heap()

```
void build_max_heap (
            std::vector< Population > & vec,
            int size,
            const std::string fileName,
            bool writeToFile )
```

Builds a binary max heap from the vector.

Parameters

vec	Vector to be built as a max heap.
size	Size of the vector.
fileName	Name of the file to be written.
writeToFile	Whether to write the output to file or not.

Time Complexity Analysis:

The `build_max_heap` function is part of the heap sort algorithm and is responsible for building a max-heap from an arbitrary array. The function iterates through the array starting from the last parent down to the root, calling `max_heapify` recursively. The time complexity of `build_max_heap` is $O(n)$, where n is the number of elements in the heap.

- **Time complexity:** The function iterates through approximately $n/2$ elements (the non-leaf nodes) and calls `max_heapify` on each. The time complexity is $O(n)$ since each call to `max_heapify` takes $O(\log n)$ time.

Real-World Usage Example:

In a real-world scenario, the `build_max_heap` function is crucial for preparing a dataset for efficient operations based on the max-heap property. For example, it could be employed in sorting algorithms like heap sort to efficiently sort a list of cities based on their populations.

- Sorting a list of cities by population: By building a max-heap on the population values of cities, the sorting process becomes more efficient, facilitating quick access to the city with the highest population.
- Priority-based processing: In a task scheduling system, using `build_max_heap` can help prioritize tasks based on certain criteria, enhancing the efficiency of task processing.

heapsort()

```
void heapsort (
            std::vector< Population > & vec,
            int size,
            const std::string fileName )
```

Sorts the vector using heapsort algorithm.

Parameters

vec	Vector to be sorted.
size	Size of the vector.
fileName	Name of the file to be written.

Time Complexity Analysis:

The `heapsort` function is an implementation of the heap sort algorithm, which involves building a max-heap and repeatedly extracting the maximum element. The time complexity of `heapsort` is $O(n \log n)$, where n is the number of elements in the heap.

- **Build max-heap:** $O(n)$ - The function calls `build_max_heap`, which takes $O(n)$ time.
- **Extract max and max_heapify:** $O(n \log n)$ - The loop iterates n times, and within each iteration, it performs a constant time swap followed by a `max_heapify` operation, which has a time complexity of $O(\log n)$.

Real-World Usage Example:

The `heapsort` function is useful for efficiently sorting a dataset in ascending order. In real-world scenarios, it can be applied to various tasks such as:

- Sorting a list of cities by population: By applying `heapsort` to the population values of cities, you can obtain a sorted list, making it easy to identify the most and least populated cities.
- Sorting tasks based on priority: In a scheduling system, tasks can be sorted based on their priority, ensuring that high-priority tasks are processed first.

max_heap_insert()

```
void max_heap_insert (
            std::vector< Population > & vec,
            std::string city,
            int key,
            const std::string fileName )
```

Inserts a new element to the heap.

Parameters

vec	Vector to be inserted.
city	City name to be inserted.
key	Population count of the city to be inserted.
fileName	Name of the file to be written.

Time Complexity Analysis:

The `max_heap_insert` function inserts a new element into a max-heap represented by a vector. The time complexity of `max_heap_insert` is $O(\log n)$, where n is the number of elements in the heap.

- **Insertion:** $O(1)$ - Pushing a new element to the end of the vector is a constant time operation.
- **Heap increase key:** $O(\log n)$ - The function calls `heap_increase_key`, which has a time complexity of $O(\log n)$.

Real-World Usage Example:

The `max_heap_insert` function is useful for dynamically updating a priority queue represented as a max-heap. In real-world scenarios, it can be applied to tasks such as:

- Adding a new city with population information: When a new city is added to the dataset, `max_heap_insert` efficiently incorporates the new information, allowing for quick access to the city with the highest population.
- Updating priority tasks: In a scheduling system, tasks can be dynamically added, and their priorities adjusted using `max_heap_insert`, ensuring that the most important tasks are processed first.

heap_extract_max()

```
Population heap_extract_max (
             std::vector< Population > & vec,
             const std::string fileName )
```

Extracts the maximum element from the heap.

Parameters

vec	Vector to be extracted.
fileName	Name of the file to be written.

Returns

The maximum element of the heap.

Time Complexity Analysis:

The `heap_extract_max` function extracts the maximum element from a max-heap represented by a vector. The time complexity of `heap_extract_max` is $O(\log n)$, where n is the number of elements in the heap.

- **Extraction and Heapify:** $O(\log n)$ - The function performs a constant time operation to retrieve the maximum element from the root, followed by a call to `max_heapify`, which has a time complexity of $O(\log n)$.

Real-World Usage Example:

The `heap_extract_max` function is useful for efficiently retrieving and processing the maximum element from a priority queue represented as a max-heap. In real-world scenarios, it can be applied to tasks such as:

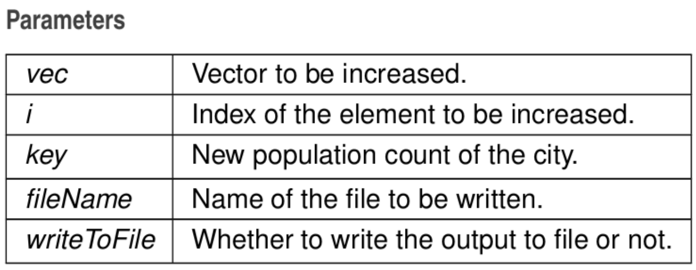
- Processing the most important task: In a scheduling system, `heap_extract_max` can be used to extract and process the highest-priority task, ensuring that critical tasks are handled promptly.

- Monitoring the most populated city: In a dataset of cities, `heap_extract_max` can efficiently retrieve and analyze the city with the highest population.

heap_increase_key()

```
void heap_increase_key (
             std::vector< Population > & vec,
             int i,
             int key,
             const std::string fileName,
             bool writeToFile )
```

Increases the key of the element at given index.



Parameters

vec	Vector to be increased.
i	Index of the element to be increased.
key	New population count of the city.
fileName	Name of the file to be written.
writeToFile	Whether to write the output to file or not.

Time Complexity Analysis:

The `heap_increase_key` function increases the key of a specific element in a max-heap represented by a vector and ensures that the max-heap property is maintained. The time complexity of `heap_increase_key` is $O(\log n)$, where n is the number of elements in the heap.

- **Updating Key and Swapping:** $O(\log n)$ - The function updates the key of a specific element and iteratively swaps it with its parent until the max-heap property is restored. The number of iterations is proportional to the height of the heap, which is $O(\log n)$.

Real-World Usage Example:

The `heap_increase_key` function is useful for dynamically updating the priority of an element in a max-heap, ensuring that the element is correctly positioned based on its new key. In real-world scenarios, it can be applied to tasks such as:

- Adjusting the priority of a city: When the population of a city changes, `heap_increase_key` can be used to efficiently update the priority in the dataset, ensuring that the city is correctly positioned in the max-heap.

- Modifying task priorities: In a scheduling system, `heap_increase_key` can be applied to adjust the priority of a task dynamically, promoting or demoting it based on changing criteria.

heap_maximum()

```
Population heap_maximum (
            std::vector< Population > & vec,
            const std::string fileName )
```

Returns the maximum element of the heap.

Parameters

vec	Vector to be searched.
fileName	Name of the file to be written.

Returns

The maximum element of the heap.

Time Complexity Analysis:

The `heap_maximum` function retrieves the maximum element from a max-heap represented by a vector. The time complexity of `heap_maximum` is $O(1)$, as it directly returns the first element of the vector.

- **Extraction:** $O(1)$ - The function directly returns the first element of the vector, which requires constant time.

Real-World Usage Example:

The `heap_maximum` function is useful for quickly obtaining the maximum element from a priority queue represented as a max-heap. In real-world scenarios, it can be applied to tasks such as:

- Identifying the city with the highest population: In a dataset of cities, `heap_maximum` can efficiently retrieve and analyze the city with the highest population.
- Retrieving the top-performing item: In an e-commerce platform, `heap_maximum` can be used to quickly identify the item with the highest performance metric.

dary_calculate_height()

```
int dary_calculate_height (
            int size,
            int d,
            const std::string fileName )
```

Calculates the height of the d-ary heap.

Parameters

size	Size of the vector.
d	Number of children of each node.
fileName	Name of the file to be written.

Returns

The height of the d-ary heap.

Time Complexity Analysis:

The `dary_calculate_height` function calculates the height of a d-ary heap based on its size and arity. The time complexity of `dary_calculate_height` is $O(1)$, as it involves constant time operations.

- **Height Calculation:** $O(1)$ - The function uses a mathematical formula to calculate the height of the d-ary heap, which takes constant time.

Real-World Usage Example:

The `dary_calculate_height` function can be utilized to determine the height of a d-ary heap, providing valuable information for managing and understanding the structure of the heap.

- Adjusting parallel processing resources: In parallel computing environments, the height of a d-ary heap can influence the allocation of resources for efficient task scheduling.
- Optimizing hierarchical data structures: The height information can be used to optimize the representation and traversal of hierarchical data structures in applications.

dary_extract_max()

```
Population dary_extract_max (
            std::vector< Population > & vec,
            int d,
            const std::string fileName )
```

Extracts the maximum element from the d-ary heap.

Parameters

vec	Vector to be extracted.
d	Number of children of each node.
fileName	Name of the file to be written.

Returns

The maximum element of the d-ary heap.

Time Complexity Analysis:

The `dary_extract_max` function extracts the maximum element from a d-ary heap represented by a vector. The time complexity of `dary_extract_max` is $O(\log_d n)$, where n is the number of elements in the heap and d is the arity of the heap.

- **Extraction and Heapify:** $O(\log_d n)$ - The function performs a constant time operation to retrieve the maximum element from the root, followed by a call to `dary_max_heapify`, which has a time complexity of $O(\log_d n)$.

Real-World Usage Example:

The `dary_extract_max` function is useful for efficiently retrieving and processing the maximum element from a priority queue represented as a d-ary heap. In real-world scenarios, it can be applied to tasks such as:

- Processing the most important task: In a scheduling system, `dary_extract_max` can be used to extract and process the highest-priority task, ensuring that critical tasks are handled promptly.
- Monitoring the most populated city: In a dataset of cities, `dary_extract_max` can efficiently retrieve and analyze the city with the highest population.

dary_insert_element()

```
void dary_insert_element (
                std::vector< Population > & vec,
                std::string city,
                int key,
                int d,
                const std::string fileName )
```

Inserts a new element to the d-ary heap.

Parameters

vec	Vector to be inserted.
city	City name to be inserted.
key	Population count of the city to be inserted.
d	Number of children of each node.
fileName	Name of the file to be written.

Time Complexity Analysis:

The `dary_insert_element` function inserts a new element into a d-ary heap represented by a vector. The time complexity of `dary_insert_element` is $O(\log_d n)$, where n is the number of elements in the heap and d is the arity of the heap.

- **Insertion and Heapify:** $O(\log_d n)$ - The function performs a constant time operation to add the new element to the end of the vector, followed by a call to `dary_increase_key`, which has a time complexity of $O(\log_d n)$.

Real-World Usage Example:

The `dary_insert_element` function is valuable for efficiently adding new elements to a priority queue represented as a d-ary heap. In real-world scenarios, it can be applied to tasks such as:

- Dynamic task scheduling: In a scheduling system, `dary_insert_element` can be used to dynamically add new tasks with varying priorities to the queue.
- Population updates: In a dataset of cities, `dary_insert_element` can efficiently add and update population data for new or growing cities.

dary_increase_key()

```
void dary_increase_key (
            std::vector< Population > & vec,
            int i,
            int key,
            int d,
            const std::string fileName,
            bool writeToFile )
```

Increases the key of the element at given index.

Parameters

vec	Vector to be increased.
i	Index of the element to be increased.
key	New population count of the city.
d	Number of children of each node.
fileName	Name of the file to be written.
writeToFile	Whether to write the output to file or not.

Time Complexity Analysis:

The `dary_increase_key` function increases the key of a specified element in a d-ary heap represented by a vector. The time complexity of `dary_increase_key` is $O(\log_d n)$, where n is the number of elements in the heap and d is the arity of the heap.

- **Key Increase and Heapify:** $O(\log_d n)$ - The function performs a constant time operation to update the key of the specified element, followed by a loop that moves the element up the heap to maintain the max-heap property. The loop has a time complexity of $O(\log_d n)$.

Real-World Usage Example:

The `dary_increase_key` function is essential for efficiently updating the priority of an element in a priority queue represented as a d-ary heap. In real-world scenarios, it can be applied to tasks such as:

- Adjusting task priorities: In a scheduling system, `dary_increase_key` can be used to increase the priority of a task, ensuring that it is processed with higher precedence.
- Modifying population data: In a dataset of cities, `dary_increase_key` can efficiently update the population of a city, reflecting changes or corrections in the data.

Additional Functions:

dary_max_heapify()

```
void dary_max_heapify (
            std::vector< Population > & vec,
            int i,
            int size,
            int d,
            const std::string fileName,
            bool writeToFile )
```

Creates the d-ary max heap structure for given index recursively.

This function is not callable from command line, it is used in dary_build_max_heap function, and creates the d-ary max heap structure for given index recursively.

Parameters

vec	Vector to be built as a max heap.
i	Index of the element to be max heapified.
size	Size of the vector.
d	Number of children of each node.
fileName	Name of the file to be written.
writeToFile	Whether to write the output to file or not.

Time Complexity Analysis:

The `dary_max_heapify` function is a modification of the `max_heapify` for d-ary heaps. It maintains the max-heap property in a d-ary heap represented by a vector. The time complexity of `dary_max_heapify` is $O(\log_d n)$, where n is the number of elements in the heap and d is the arity of the heap.

- **Index Calculation:** $O(d)$ - The function calculates the indices of the children of a node in the d-ary heap, which takes $O(d)$ time.
- **Finding Largest Child:** $O(d)$ - The function iterates over the children's indices and finds the largest child, taking $O(d)$ time.
- **Heapify Operation:** $O(\log_d n)$ - The function performs the heapify operation by recursively calling itself, and the time complexity is $O(\log_d n)$.

Real-World Usage Example:

The `dary_max_heapify` function is crucial for maintaining the max-heap property in d-ary heaps, which can be used in various real-world scenarios:

- Organizing hierarchical data: In applications where data has a natural hierarchical structure with a variable number of dependencies, a d-ary heap can efficiently represent this hierarchy.
- Task scheduling in parallel systems: D-ary heaps can be employed in parallel systems to efficiently schedule and manage tasks with dependencies.

dary_build_max_heap()

```
void dary_build_max_heap (
            std::vector< Population > & vec,
            int size,
            int d,
            const std::string fileName,
            bool writeToFile )
```

Builds a d-ary max heap from the vector.

This function is not callable from command line, it is used in building d-ary heaps before specific function calls.

Parameters

vec	Vector to be built as a max heap.
size	Size of the vector.
d	Number of children of each node.
fileName	Name of the file to be written.
writeToFile	Whether to write the output to file or not.

Time Complexity Analysis:

The `dary_build_max_heap` function is part of the heap sort algorithm and is responsible for building a max-heap from an arbitrary array in a d-ary heap. The function iterates through the array starting from the last parent down to the root, calling `dary_max_heapify` recursively. The time complexity of `dary_build_max_heap` is $O(d \cdot n)$, where n is the number of elements in the heap and d is the arity of the heap.

- **Time Complexity:** $O(d \cdot n)$ - The function iterates through approximately n/d elements (the non-leaf nodes) and calls `dary_max_heapify` on each. The time complexity is $O(d \cdot n)$ since each call to `dary_max_heapify` takes $O(\log_d n)$ time.

Real-World Usage Example:

In a real-world scenario, the `dary_build_max_heap` function is crucial for preparing a dataset for efficient operations based on the max-heap property in a d-ary heap. For example, it could be employed in sorting algorithms like heap sort to efficiently sort a list of cities based on their populations in a d-ary heap.

- **Sorting a List of Cities by Population in a D-ary Heap:** By building a max-heap on the population values of cities in a d-ary heap, the sorting process becomes more efficient, facilitating quick access to the city with the highest population.

- **Priority-Based Processing in a D-ary Heap:** In a task scheduling system utilizing a d-ary heap, using `dary_build_max_heap` can help prioritize tasks based on certain criteria, enhancing the efficiency of task processing.

dary_heapsort()

```
void dary_heapsort (
            std::vector< Population > & vec,
            int size,
            int d,
            const std::string fileName )
```

Sorts the vector using heapsort algorithm.

This function is not callable from command line, it is used for sorting d-ary heaps for testing purposes.

Parameters

vec	Vector to be sorted.
size	Size of the vector.
d	Number of children of each node.
fileName	Name of the file to be written.

Time Complexity Analysis:

The `dary_heapsort` function is part of the heap sort algorithm for a d-ary heap. It first builds the max-heap using `dary_build_max_heap` and then iteratively swaps the maximum element with the last element while calling `dary_max_heapify` recursively. The time complexity of `dary_heapsort` is $O(n \cdot \log_d n)$, where n is the number of elements in the heap and d is the arity of the heap.

- **Build Max Heap:** $O(d \cdot n)$ - The function calls `dary_build_max_heap`, which has a time complexity of $O(d \cdot n)$.

- **Iterative Heapify:** $O(n \cdot \log_d n)$ - The function iteratively calls `dary_max_heapify` and performs swaps, resulting in a time complexity of $O(n \cdot \log_d n)$.

Real-World Usage Example:

In a real-world scenario, the `dary_heapsort` function can be employed to efficiently sort a list of cities based on their populations in a d-ary heap.

- **Efficient Sorting in a D-ary Heap:** By utilizing the d-ary heap structure, `dary_heapsort` efficiently sorts a list of cities based on their populations. This is particularly beneficial when dealing with datasets where a d-ary heap representation is suitable.

- **Priority-Based Processing in a D-ary Heap:** The sorted list can be used for priority-based processing, such as identifying the most populated cities or managing tasks in a scheduling system with dependencies.

1.3. Comparative Analysis of the Heapsort Algorithm with the Quicksort Algorithm

Elapsed Time Comparison

Execution times in nanoseconds (ns) with different pivoting strategies of Quicksort and Heapsort.

	Population1	Population2	Population3	Population4
Heapsort	15.700.117	16.993.899	16.234.436	17.011.210
Last Element	86.672.552	320.283.823	339.709.786	8.619.671
Random Element	7.243.895	2.411.919	7.526.918	8.615.182
Median of 3	8.617.122	2.607.245	6.959.057	9.074.652

Table 1.1: Execution times of different pivoting strategies of Quicksort and Heapsort on input data.

Population1 (Mostly Descending Order):

- **Quicksort vs. Heapsort:**
  - Quicksort might exhibit better performance due to its efficiency in handling partially sorted or nearly sorted datasets.
  - Heapsort, requiring more comparisons, could lead to a longer elapsed time.

Population2 (Ascending Order):

- **Quicksort vs. Heapsort:**
  - Quicksort with the last element pivoting strategy might struggle with ascending order, potentially leading to more comparisons and increased elapsed time.
  - Heapsort, with its consistent performance, could provide a more stable runtime.

Population3 (Descending Order):

- **Quicksort vs. Heapsort:**
  - Quicksort might perform relatively well, as the descending order allows for efficient partitioning.
  - Heapsort, although more comparisons are needed, may still deliver a competitive runtime.

Population4 (Unsorted):

- **Quicksort vs. Heapsort:**
  - Both algorithms might face challenges with an unsorted dataset.

    - Quicksort, depending on the pivot strategy, may vary in performance but gives a stable output on average.
    - Heapsort's consistent nature may provide stable performance.

Number of Comparisons

Number of comparisons made by the different pivoting strategies of naiveQuickSort and Heapsort.

	Population1	Population2	Population3	Population4
Heapsort	876.598	984.193	886.253	937.218
Last Element	21.724.019	177.063.113	186.240.040	594.736
Random Element	544.015	576.351	573.219	536.574
Median of 3	554.964	567.906	558.304	545.023

Table 1.2: Number of comparisons of different pivoting strategies of Quicksort and Heapsort on input data.

The number of comparisons can vary based on the pivoting strategy, and it can be anticipated that the heapsort algorithm may involve more comparisons compared to quicksort because heapsort inherently relies on a larger number of comparisons. This tendency becomes more evident when examining the results for Population4. Observing the outcomes, it can be noted that all pivot strategies in quicksort produce similar outputs, and all of them outperform heapsort. Here, it can be inferred that heapsort is more dependent on the number of comparisons, resulting in a slightly longer execution time compared to quicksort.

Heapsort Comparisons

- Generally consistent across different populations.
- Performs a moderate number of comparisons, with the count relatively stable regardless of the input order.

Quicksort Comparisons

- **Last Element Pivot:**
    - Extremely high comparisons, especially noticeable in Population2.
    - Performs poorly when the data is already sorted or nearly sorted.
- **Random Element Pivot:**
    - Performs better than the last element but still high.
    - Less affected by sorted input data.
- **Median of 3 Pivot:**

  - Shows improvement compared to random and last element pivots.
  - More stable performance across different input orders.

Overall Comparison of Heapsort and Quicksort

- **Heapsort:** Consistently moderate comparisons across various input orders.
- **Quicksort:** Highly dependent on the choice of the pivot.
  - Last Element: Poor performance on nearly sorted data.
  - Random Element: More robust but still relatively high.
  - Median of 3: Improved stability, better than the other pivot strategies.

In summary, Heapsort demonstrates more consistent behavior across different datasets, while Quicksort's performance varies significantly based on the choice of pivot. The "Median of 3" strategy in Quicksort improves its behavior, making it more competitive with Heapsort in various scenarios.

Insights into Strengths and Weaknesses

Quicksort:

- **Strengths:**
  - Efficient in handling partially sorted datasets.
  - Pivot selection strategies can influence performance.
- **Weaknesses:**
  - Sensitive to input order, especially in ascending or descending sequences.

Heapsort:

- **Strengths:**
  - Consistent performance in various scenarios.
  - Well-suited for unsorted datasets.
- **Weaknesses:**
  - Generally involves more comparisons.
  - May not outperform quicksort in certain partially sorted scenarios.

Scenarios

- **Quicksort Preference:**
  - Partially sorted datasets, especially in descending order.

    - Input datasets with characteristics favoring efficient pivot selection.

- **Heapsort Preference:**
    - Unsorted datasets where consistent performance is crucial.
    - Situations where the number of comparisons is not a critical factor.

In conclusion, the choice between heapsort and quicksort depends on the characteristics of the input data. Quicksort may excel in specific scenarios, but heapsort offers more consistent performance across various input types, especially when the dataset is unsorted.