A COMPARATIVE ANALYSIS OF HEAPSORT AND COCKTAIL SORT IN QUICKTICKET: A MOVIE TICKET BOOKING SYSTEM

Jynn Diane Casili[1], John Augustus Escobin[2], Arren Lei Mendoza[3], Alejandro Matute[4], Maribel Cardona[5], Sherwin Sapin[6], Jefferson Lerios[7], Loyd Echalar[8]

Laguna State Polytechnic University – Los Baños Campus, Los Baños, Laguna

casilijynn@gmail.com[1], ajescobin333@gmail.com[2], arrenleimendoza04@gmail.com[3], alejandroventiromatute@lspu.edu.ph[4], mvcardona57@gmail.com[5], sapiin.sherwin@gmail.com[6], jeffersonlerios@yahoo.com[7], loydechalar@lspu.edu.ph[8]

Abstract

This study investigated the performance of two sorting algorithms, Heapsort and Cocktail Sort, within the context of the QuickTicket system, a ticket-booking platform. The main objective was to compare the runtime efficiency and scalability of these algorithms with varying dataset sizes, specifically focusing on sorting ticket information. Through performance testing, the research demonstrates that Heapsort outperforms Cocktail Sort across all dataset sizes, exhibiting significantly better scalability and runtime efficiency. Heapsort consistently processed datasets of 100, 300, and 500 tickets in a fraction of the time compared to Cocktail Sort, which showed increasing inefficiency as dataset size grew. Additionally, unit testing confirmed the functionality and reliability of key components within the QuickTicket system, including ticket booking, cart management, and sorting algorithms. The findings highlight the importance of algorithmic efficiency in maintaining system performance, particularly for real-time applications handling large volumes of data. The study concludes that Heapsort is the optimal choice for the QuickTicket system, ensuring faster processing times and a more reliable user experience.

I. Introduction

This chapter provides an overview of the study, including the project context, objectives, purpose, and the scope and limitations. It establishes the foundation for understanding the relevance and significance of the research, particularly in optimizing real-time data management within a movie ticket booking system. Additionally, it outlines the key concepts and challenges addressed, setting the stage for a detailed exploration of Heapsort and Cocktail Sort as sorting algorithms.

Sorting algorithms play a critical role in computer science, particularly in organizing and managing data efficiently (Cormen et al., 2009). In the context of movie ticket booking systems, effective sorting mechanisms are essential to handle real-time updates and large datasets, ensuring seamless user experiences even during peak demand. This study centers on the integration of Heapsort and Cocktail Sort within the QuickTicket: Movie Ticket Booking System. QuickTicket is designed to manage ticket organization, genre-based filtering, and cart management by utilizing Heapsort and Cocktail Sort. These algorithms were chosen for their distinct advantages in speed, scalability, and memory usage. While Heapsort leverages the efficiency of heap data structures to excel at large, unsorted datasets (William, 1964), Cocktail Sort offers advantages in managing smaller or nearly sorted datasets due to its bidirectional sorting mechanism (Knuth, 1997).

The project emphasizes the optimization of ticket sorting, providing insights into algorithm selection based on data characteristics, and ultimately improving user experience during ticket booking processes (Sedgewick & Wayne, 2011).

The primary objectives of this study are:

1. Compare the runtime performance of Heapsort and Cocktail Sort for a dataset size of 100 tickets.

2. Compare the runtime performance of Heapsort and Cocktail Sort for a dataset size of 300 tickets.
3. Compare the runtime performance of Heapsort and Cocktail Sort for a dataset size of 500 tickets.
4. Assess the scalability of both algorithms as the dataset size increases.

This research aims to provide practical insights into the application of sorting algorithms for real-time system. By comparing Heapsort and Cocktail Sort in QuickTicket, this study seeks to optimize the ticket booking process. The findings will guide developers in selecting suitable algorithms, ensuring fast, accurate, and efficient operations even under high-demand conditions.

The Statement of the Problem of this study are:

1. What is the runtime performance of Heapsort and Cocktail Sort when sorting a dataset of 100 tickets?
2. How do Heapsort and Cocktail Sort compare in terms of runtime performance for a dataset of 300 tickets?
3. How do Heapsort and Cocktail Sort perform when sorting a dataset of 500 tickets?
4. How scalable are Heapsort and Cocktail Sort as the dataset size increases?

The study focuses on quantitatively analyzing the performance of Heapsort and Cocktail Sort using simulated datasets of varying sizes (100, 300, and 500 tickets) in the QuickTicket system. The research investigates runtime complexity, resource utilizations, and scalability across different dataset conditions.

This study is conducted in a simulated environment and does not involve real-world user testing or a fully functional movie ticket booking system. It uses randomly generated data, which may not fully represent real-world booking scenarios. Additionally, the analysis excludes other factors such as user interaction, stability, and external system constraints.

This study is significant for the following beneficiaries:

For **System Developers,** the findings are significant as they provide valuable insights into selecting the most efficient sorting algorithm for real-time applications. By applying this knowledge, developers can enhance the performance of ticket-booking systems, ensuring smoother and faster operations.

For **Computer Science Researchers and Students,** this study offers a deeper understanding of sorting algorithms, their performance, and their applications in real-world scenarios. It serves as a comparative reference for future research, aiding students and researchers in exploring and optimizing algorithms further.

For **Businesses Using Ticket-Booking Systems,** this study is significant as it ensures faster and more efficient processing of large datasets, resulting in better customer satisfaction. Moreover, it demonstrates how the thoughtful selection of algorithms can improve operational reliability, particularly during periods of high demand.

For **Educational Institutions,** the study contributes to the educational field by enriching the knowledge base on sorting algorithms. It highlights their advantages and limitations, offering students and educators a clearer perspective on their practical applications.

II. Review of Related Literature

The research reviewed highlights the strengths of Heapsort and Cocktail Sort in different contexts. Heapsort is known for its efficiency with large datasets, offering consistent O(n log n) time complexity and minimal memory usage, making it ideal for resource-constrained environments, as shown by Williams (1964) and Gupta et al. (2017). Research by Bojesen et al. (2000) and Wand and Wu (2003) further emphasize that Heapsort's performance can be enhanced through memory optimizations and platform-specific adjustments.

Cocktail sort, while less efficient than Heapsort, excels with nearly sorted data due to its two-way movement sorting approach. Studies by Elkahlout and Maghari (2003) and Ozdemir and Yilmaz (2018) highlight its usefulness in educational settings for demonstrating sorting concepts, with recent work by Ahmad and Rauf (2020) showing its adaptability in hybrid algorithms.

These findings directly inform this study by providing a clear understanding of the advantages and limitations of both algorithms. The research aims to explore how these sorting algorithms can be optimized for practical applications, building on existing knowledge to improve efficiency and performance.

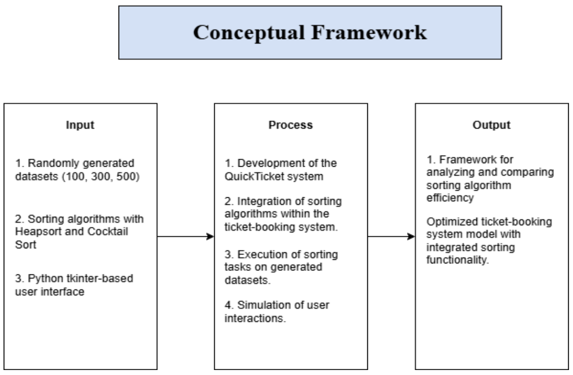

Conceptual Framework
Input
1. Randomly generated datasets (100, 300, 500)
2. Sorting algorithms with Heapsort and Cocktail Sort
3. Python tkinter-based user interface
Process
1. Development of the QuickTicket system
2. Integration of sorting algorithms within the ticket-booking system.
3. Execution of sorting tasks on generated datasets.
4. Simulation of user interactions.
Output
1. Framework for analyzing and comparing sorting algorithm efficiency
Optimized ticket-booking system model with integrated sorting functionality.

Figure 1: Conceptual Framework

This study presents a conceptual framework for analyzing the efficiency of sorting algorithms within a ticket-booking system. The input consists of randomly generated datasets of varying sizes (100, 300, and 500 records), two sorting algorithms (Heapsort and Cocktail Sort), and a Python Tkinter-based user interface. The process involves developing the QuickTicket system, integrating the sorting algorithms into the ticket-booking workflow, executing sorting tasks on the datasets, and simulating user interactions to assess performance. The output includes a framework for comparing the efficiency of the sorting algorithms and an optimized ticket-booking system model that incorporates sorting functionality for improved user experience and system performance.

III. Methodology

The methodology used is experimental method to compare the performance of Heapsort and Cocktail Sort. Both sorting algorithms were applied to randomly generated datasets of tickets, with their runtimes measured and compared across different dataset sizes of 100, 300, and 500 tickets. Each test was repeated three times to ensure accuracy and reliability. The collected data was analyzed by calculating the average runtime for each algorithm, allowing for a fair and accurate comparison of their performance.
The QuickTicket is designed to handle movie ticket booking and sorting using two algorithms: Heapsort and Cocktail Sort. The system first allows users to book tickets. After booking, the system generates a dataset of tickets, which can be sorted using either Heapsort or Cocktail Sort. The runtime for both algorithms is measured and logged, and this data is then compared to evaluate the algorithms' efficiencies.
Key components of the system include:

Ticket Booking Module: Users select movies, showtimes, and book tickets.
Sorting Module for Heapsort: This module sorts the tickets using the Heapsort algorithm and records runtime.
Sorting Module for Cocktail Sort: Similar to the Heapsort module, this one sort tickets using the Cocktail Sort algorithm and measures runtime.
Runtime Measurement Module: This component tracks and logs the runtime of both sorting algorithms, ensuring accurate data collection.

The software development life cycle (SDLC) followed in this project consists of the following phases

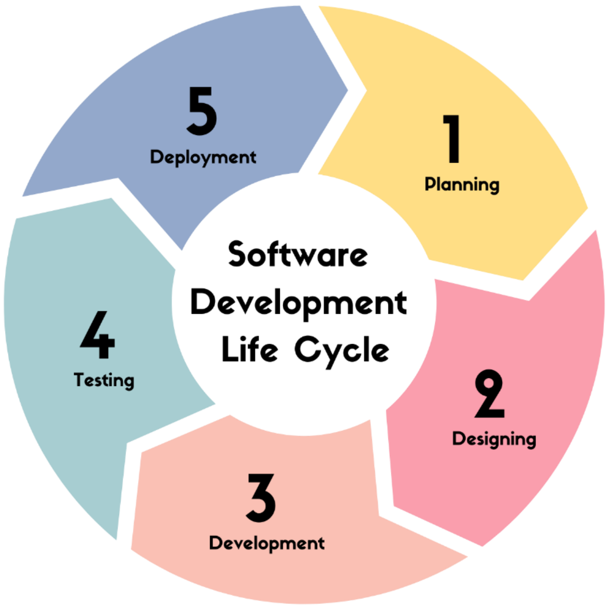


Figure 2: SDLC Model

The **planning phase** involved identifying the requirements of the QuickTicket system and establishing the need to compare two sorting algorithms: Heapsort and Cocktail Sort. This phase focused on defining the objectives and determining the most effective approach to optimize the ticket-booking process.

During the **design phase**, the structure of the system was outlined. Figma was used to create visual representations of key features, including the ticket booking modules and sorting functions. The methods for data collection and performance analysis were carefully planned to ensure accurate results.

The **development phase** focused on implementing the QuickTicket system. Python was used to integrate Heapsort and Cocktail Sort algorithms into the system. Functions for measuring and logging runtime performance were also developed to support the comparison of the two algorithms.

Testing phase involved using sample datasets with sizes of 100, 300, and 500 tickets. These tests were conducted to evaluate the runtime performance and scalability of Heapsort and Cocktail Sort. The results provided insights into the efficiency of each algorithm under varying dataset sizes.

Lastly, in the **deployment phase**, the system was finalized and made operational. This allowed for the practical analysis of the sorting algorithms, ensuring that the system could effectively handle ticket booking and sorting tasks in a real-world setting.

The products testing and evaluation focused on ensuring the reliability and accuracy of the QuickTicket system, particularly in the implementation of Heapsort and Cocktail Sort algorithms. Functional testing verified that all features, including ticket booking, sorting, and runtime measurement, worked as intended. Integration testing ensured seamless interaction between modules, with proper data flow between the booking and sorting functions. The system's interface, developed with Python Tkinter, was evaluated for usability, gathering feedback on navigation, layout, and user interactions to ensure a satisfactory experience.

Performance testing aimed to evaluate the efficiency of Heapsort and Cocktail Sort in sorting datasets of varying sizes within the QuickTicket system. Datasets of 100, 300, and 500 tickets were processed, with the runtime of each sorting algorithm measured three times to ensure consistency. The tests were conducted under controlled conditions to minimize any external factors, such as processing power of system load, that could influence the results.

To assess the scalability of the algorithms, datasets of increasing sizes were tested to observe how well the runtime held up against the expected computational complexity. This provided valuable insights into the system's ability to manage larger datasets in real-world conditions. The tests were limited to Heapsort and Cocktail sort, and comparisons with other algorithms were suggested as a potential area for future research.

Data was collected by running both Heapsort and Cocktail Sort on datasets of 100, 300, and 500 tickets, generated using a custom function in the QuickTicket system. Each dataset was sorted three time to minimize variations from factors like system load. The runtime for both algorithms was measured and logged systematically for easy comparison. While the analysis compared the average runtime of Heapsort and Cocktail Sort for each dataset size. The runtime values from three trials were averaged to ensure reliable performance data. This allowed for an assessment of each algorithm's efficiency, scalability, and performance trends as dataset sizes increased, highlighting the strengths and weaknesses of both.

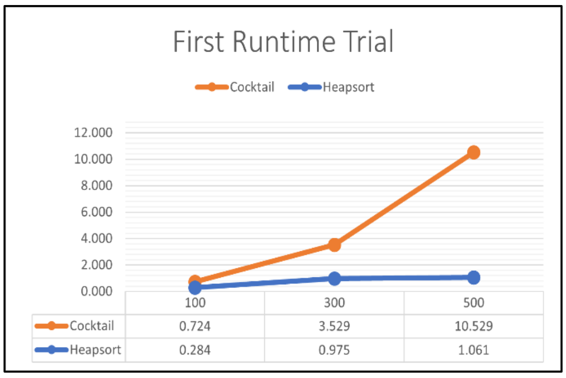
IV: Results and Discussion

The primary goal of the project was to assess and compare the performance of Heapsort and Cocktail Sort algorithms in terms of runtime efficiency with datasets of varying sizes. These objectives guided the analysis of the results and the selection of the most suitable sorting algorithm for the QuickTicket system. The results are summarized in tables and graphs for clarity, followed by an analysis for each objective and trial

Sorting Algorithm	Average Runtime
Heapsort	0.287
Cocktail Sort	0.658

Table 1: Runtime Comparison for 100 Tickets

For datasets consisting 100 tickets, Heapsort completed the sorting process in 0.287 ms, whereas Cocktail Sort required 0.658 ms. This demonstrates Heapsort's efficiency, even for smaller datasets, confirming its suitability for quick sorting tasks. Cocktail Sort, while slower, remains functional but less optimal for such dataset sizes.



	100	300	500
Cocktail	0.724	3.529	10.529
Heapsort	0.284	0.975	1.061

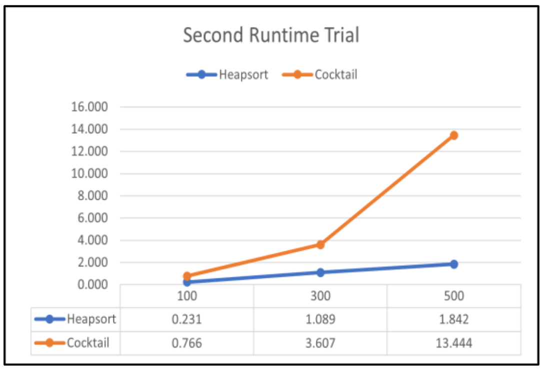
Graph 1: First Runtime Trial

The first runtime trial shows a clear difference between the performance of Heapsort and Cocktail Sort. Heapsort consistently exhibits a shorter runtime for all dataset sizes, reflecting its efficient use of the heap data structure. Cocktail Sort, on the other hand, demonstrates higher runtimes, particularly for larger datasets, as its bidirectional sorting mechanism becomes less effective.

Sorting Algorithm	Average Runtime
Heapsort	0.970
Cocktail Sort	4.433

Table 2: Runtime Comparison for 300 Tickets

As the dataset size increased to 300 tickets, the performance gap between the two algorithms widened. Heapsort processed the dataset in 0.970 ms, maintaining its scalability and efficiency. In contrast, Cocktail Sort's runtime rose significantly to 4.433 ms, indicating its growing inefficiency with larger datasets.



	100	300	500
Heapsort	0.231	1.089	1.842
Cocktail	0.766	3.607	13.444

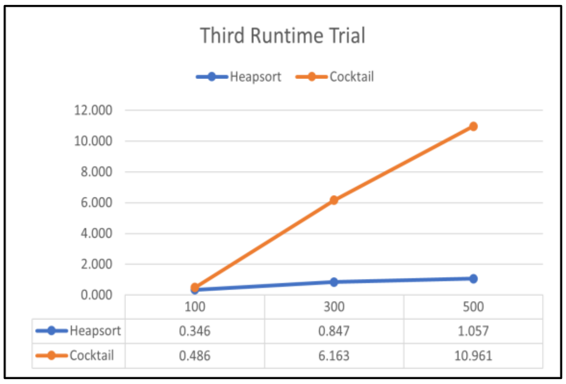
Graph 2: Second Runtime Trial

In the second trial, the trend remains consistent. Heapsort continues to outperform Cocktail Sort across all dataset sizes, with only a slight variation in runtime compared to the first trial. This consistency highlights Heapsort's reliability in maintaining efficient performance regardless of dataset size. Cocktail Sort's runtime once again increases significantly as the dataset size grows, emphasizing its inefficiency for handling larger datasets.

Sorting Algorithm	Average Runtime
Heapsort	1.320
Cocktail Sort	11.644

Table 3: Runtime Comparison for 500 Tickets

For the largest dataset size of 500 tickets, Heapsort maintained a stable performance with a runtime of 1.320 ms, while Cocktail Sort required a much longer time of 11.644 ms. This difference highlights Heapsort's ability to handle large datasets effectively, while Cocktail Sort struggles with increased data sizes due to its less efficient algorithmic structure.



	100	300	500
Heapsort	0.346	0.847	1.057
Cocktail	0.486	6.163	10.961

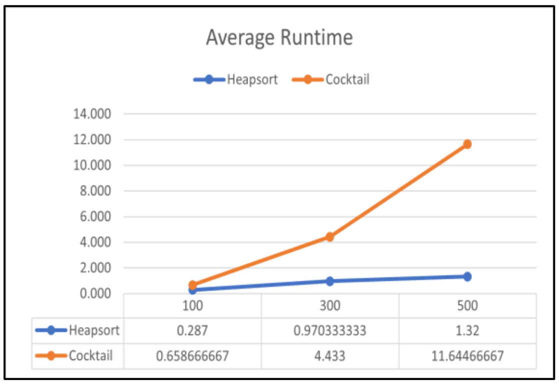
Graph 3: Third Runtime Trial

The third runtime trial further solidifies the pattern observed in the previous trials. Heapsort maintains its low runtime across all dataset sizes, while Cocktail Sort exhibits a dramatic increase in runtime for larger datasets. This trial reinforces the conclusion that Heapsort is better suited for systems like QuickTicket, where scalability and quick processing are essential.

Dataset Size	Heapsort Runtime	Cocktail Sort Runtime
100	0.287	0.658
300	0.970	4.433
500	1.320	11.644

Table 4: Scalability Comparison Across Dataset Sizes

The scalability of Heapsort and Cocktail Sort is evident from the results summarized in Table 4. Heapsort demonstrated a consistent and gradual increase in runtime as the dataset size grew, reflecting its efficient O(n log n) complexity. On the other hand, Cocktail Sort exhibited a sharp rise in runtime, consistent with its O(n^2) complexity, making it unsuitable for larger datasets.



	100	300	500
Heapsort	0.287	0.970333333	1.32
Cocktail	0.658666667	4.433	11.64466667

Graph 4: Average Runtime Trial

Unit tests were performed on several key features of the system, including ticket booking, cart management, movie selection, and sorting algorithms. The objective of the unit tests was to ensure that each feature operates as intended and meets the functional requirements of the system in which all results showed that each component is working as intended.

In addition to individual feature testing, the entire system was tested as a whole to ensure that all components worked together seamlessly. This included end-to-end testing of the ticket booking process, cart managements, genre and movie selection, sorting functionalities, and ticket generation. The system passed the overall integration tests without any significant issues. All components worked together as expected, and users were able to perform the entire ticket booking and management process without errors. The integration tests confirmed that the system functions smoothly and meets the requirements for functionality.

The analysis focuses on identifying performance trends and explaining the observed differences. Heapsort consistently showed smaller runtimes and better scalability across all dataset sizes. For example, it processed 100 tickets in 0.287 ms, 300 tickets in 0.970 ms, and 500 tickets in 1.320 ms, demonstrating a steady increase in runtime with larger datasets.

In contrast, Cocktail Sort struggled with larger datasets, taking 0.658 ms for 100 tickets, 4.433 ms for 300 tickets, and 11.644 ms for 500 tickets. Its runtime increased sharply with dataset size, showing inefficiency in scaling. While the performance gap was minimal for smaller datasets, Heapsort became three to eight times faster as dataset sizes grew, solidifying its superiority in runtime efficiency.

V. Summary, Conclusion, and Recommendation

The study focused on evaluating the efficiency of sorting algorithms within the QuickTicket system, specifically comparing Heapsort and Cocktail Sort in terms of runtime performance.

Heapsort demonstrated superior runtime efficiency, successfully handling large datasets and scaling effectively in high-demand scenarios. While Cocktail Sort performed significantly worse with larger datasets, making it unsuitable for systems requiring high scalability. The study highlighted the importance of efficient sorting algorithms to maintain responsiveness and reliability in real-time ticket-booking systems.

In conclusion, the research confirmed the crucial role of selecting the right algorithm in optimizing system performance. Heapsort was found to be the optimal sorting algorithm for the QuickTicket system due to its speed and scalability, outperforming Cocktail Sort in all tested scenarios.

By integrating Heapsort into the ticket-booking system, users can benefit from improved processing times and an enhanced experience, even under high system loads. The findings underscore the value of thoughtful algorithm design in ensuring the reliability and efficiency of systems that process large volumes of data.

The study offers the following recommendations:

1. Adopt Heapsort as the primary sorting mechanism for QuickTicket to improve system responsiveness and user satisfaction.

2. Conduct real-world load testing to validate Heapsort's performance and ensure it meets user expectations.

3. Look into combining Heapsort with other algorithms to see if it can work

even faster or handle unique scenarios better.

4. Pay attention to how sorting speed affects users. A system that feels smooth and fast can make a big difference in how happy people are with it.

REFERENCES

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT press.

Knuth, D. E. (1997). The art of computer programming, volume 3: Sorting and searching (2nd ed.). Addison-Wesley Professional.

Sedgewick, R., & Wayne, K. (2011). Algorithms. Addison-Wesley Professional.

Williams, J. W. J. (1964). Algorithm 232—Heapsort. Communications of the ACM, 7(6), 347–348.

Cormen T. H., *Introduction to Algorithms*, 2009, MIT press, Cambridge, MA, USA.

Elkahlout A. H. and Maghari A. Y. A., *A comparative study of sorting algorithms comb, cocktail and counting sorting*, 2017.

Hammad J., A comparative study between various sorting algorithms, *International Journal of Computer Science and Network Security (IJCSNS)*. (2015) 15, no. 3.

Al-Kharabsheh K. S., AlTurani I. M., AlTurani A. M. I., and Zanoon N. I., Review on sorting algorithms a comparative study, *International Journal of Computer Science and Security (IJCSS)*. (2013) 7, no. 3, 120–126.

Sharma, V. (2009). *Comparative performance study of improved heap sort algorithm on different hardware.* Journal of Computer Science, 5(7), 476-478.

Kulalvaimozhi, V.P., et al. (2021). *Performance analysis of Heap Sort and Insertion Sort Algorithm*. International Journal of Emerging Trends in Engineering Research, 9(5).

Hulín, M. (2011). *A detailed analysis of sorting algorithms: Stability, pseudo-code, and complexity analysis*. 9(5).

Lakshmi, Dr. (2014). *Time complexity analysis of sorting algorithms: Quick Sort, Insertion Sort, Heap Sort, and Merge Sort in a C# environment*. International Journal of Emerging Trends in Engineering Research 9(5)

Jehad A. and Rami M., An enhancement of major sorting algorithms, *International Arab Journal of Information Technology*. (2010) 7, no. 1.