Module 4 Milestone 3

Hai Nguyen

CS 499 Computer Science Capstone

Professor Brook Goggin

17 Nov 2023

## Algorithms and Data Structures:

The artifact "vectorSorting.cpp" is a program created by Hai Nguyen, a Computer Science major student at SNHU. This program serves as a collection of bids, implementing two distinct sorting algorithms: selection sort and quicksort. In addition to the sorting functionalities, the program includes features for loading bid data from a CSV file, displaying the bids, and timing the sorting operations. The creation date of this artifact is December 4, 2021.

## 2. Justify the inclusion of the artifact in my ePorfolio:

The inclusion of the "vectorSorting.cpp" artifact in my ePortfolio is justified by its alignment with Course Outcome 1, which emphasizes proficiency in sorting algorithms. This artifact stands out as a demonstration of my skills and abilities in the field of algorithm implementation and optimization. The artifact showcases my proficiency in sorting algorithms, with a specific focus on quicksort. This is crucial for a computer science professional as it reflects a solid understanding of fundamental algorithms, which is essential in various software development scenarios.

The enhancement plan identifies that the artifact aligns with Course Outcome 1. It explicitly addresses the importance of implementing, optimizing sorting algorithms, and emphasizing the potential impact on computer system and application performance. The enhancement plan suggests that optimizing sorting algorithms can support diverse audiences in making informed decisions by enhancing the performance of computer systems and applications. This indicates the broader significance of the artifact in practical applications.

The artifact demonstrates my ability to implement quicksort, a complex sorting algorithm. This involves understanding the underlying principles of the algorithm and translating that understanding into functional and efficient code. Additionally, the functionality to load bid data from a CSV file and display bids showcases my skills in file handling, data manipulation, and creating user-friendly interfaces. These components contribute to the overall usability of the program. Overall, the inclusion of timing mechanisms in the artifact demonstrates my awareness of performance considerations. This aspect showcases my ability to evaluate and optimize algorithms, emphasizing not only correctness but also efficiency.

The enhancement plan recommends further optimization and evaluation of sorting algorithms, particularly quicksort, to improve their efficiency and performance. This improvement could involve fine-tuning the existing quicksort implementation, exploring variations of the algorithm, or even experimenting with alternative sorting algorithms to achieve better overall performance.

## 3. Course expectation:

I believe I have met the planned course objectives through the enhancements made in Module One. At this point, my current outcome-coverage plans remain comprehensive. The "vectorSorting.cpp" artifact is included in my ePortfolio because it effectively demonstrates my proficiency in sorting algorithms, aligns with relevant course outcomes, and showcases specific components highlighting my skills in software development.

4. Reflect on the process of enhancing and/or modifying the artifact:

During the creation of the artifact of vectorSorting.cpp, I would understand advanced sorting algorithms, particularly quicksort. Implementing quicksort requires a nuanced grasp of algorithmic principles and the ability to translate that understanding into functional code. Besides, the enhancement process provided an opportunity to delve into the optimization of sorting algorithms. I learned techniques to improve the efficiency and performance of the existing quicksort implementation, exploring ways to fine-tune and optimize the algorithm.

There were some challenges that I faced. Implementing and optimizing quicksort presented challenges due to its fundamental complexity. Balancing the need for an efficient algorithm with the intricacies of the implementation required careful consideration and problem-solving. Additionally, ensuring the correctness of the sorting algorithms, especially after modifications, involved rigorous debugging and testing. Identifying and addressing issues in the code required attention to detail and a systematic debugging process.

In summary, the process of enhancing and modifying the "vectorSorting.cpp" artifact provided a rich learning experience, encompassing advanced algorithm implementation, performance optimization, real-world application development, and the challenges associated with algorithmic complexity.

## Original Code: ClickedItemActivity.java:

void quickSort(vector<Bid>& bids, int begin, int end) {

int mid = 0;

//if zero or one bid to sort, then done

if (begin >=end) {

return;

}

// partition bids into low and high parts

mid = partition(bids, begin, end);

//recursively call quicksort using midpoint value (begin to end)

quickSort(bids, begin, end);

//recursively call quicksort using midpoint value (mid + 1, end)

quickSort(bids, mid + 1, end);

}

## 4.2 Modified Code: vectorSortingModified.java.

• Modified code: Switch to a different sorting algorithm, insertion sort when the partition size becomes small (< 10). The idea is that insertion sort can be more efficient than quicksort for small datasets due to its lower overhead. Overall, this optimization aims to reduce the overhead of recursive calls in quicksort for small partitions, contributing to better performance in scenarios where quicksort may be less efficient due to the associated function call overhead.

void quickSort(vector<Bid>& bids, int begin, int end) {

int mid = 0;

// Optimization: Use insertion sort for small partitions

if (end - begin + 1 <= 10) {

// Insertion sort implementation

for (int i = begin + 1; i <= end; ++i) {

Bid key = bids[i];

int j = i - 1;

// Move elements of bids[begin..i-1] that are greater than key.title

// to one position ahead of their current position

while (j >= begin && bids[j].title.compare(key.title) > 0) {

bids[j + 1] = bids[j];

--j;

}

// Insert the key into the appropriate position

bids[j + 1] = key;

}

} else {

// Continue with quicksort for larger partitions

mid = partition(bids, begin, end);

quickSort(bids, begin, end);

quickSort(bids, mid + 1, end);

}