CSC 345 Project

Hybrid Sorting Algorithms

Presented to you by:
Eiza S, Ethan W, Angelina A, Hayden R

Table of Contents

Introduction

In class, we covered numerous common sorting algorithms such as Insertion Sort or Heap Sort, but we didn't spend much time covering ***hybrid*** sorting algorithms that each emphasize the **strengths** of two sorting algorithms while minimizing the impact of their **weaknesses**.

- The theme of our hybrid algorithms is **Merge Sort**, where we combined it with five other major sorting algorithms (not including Merge Sort itself) — Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, and Quick Sort.
- We also generalized the math for the runtime and space complexity for the algorithms (see slides 18-22).

To help showcase these hybrid algorithms, we also designed a Java program that has two main goals: **(1) visualize a sorting algorithm with a bar graph of elements in an input array** and **(2) plot the total access count of a sorting algorithm as the size of the input array (N) changes**. We will feature both of these main features as we discuss each sorting algorithm.

Overall, our goal is to demonstrate that Merge Sort is an effective way of vastly improving the regular runtime of the standalone algorithms as well as the space complexity of Merge Sort.

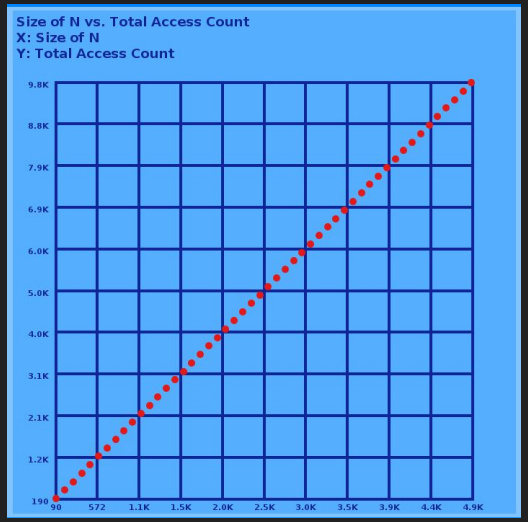
Bubble Sort: Overview

- This algorithm uses nested loops to iterate through an array and compare adjacent elements – swapping them if they are unsorted.
- The outer loop will continue looping until there are no more swaps left to perform, and the inner loop iterates over the entire array, performing swaps where possible.
- Since Bubble Sort is an in-place sorting algorithm, its space complexity is **O(1)**.

Bubble Sort: Runtimes on Access Count Plots

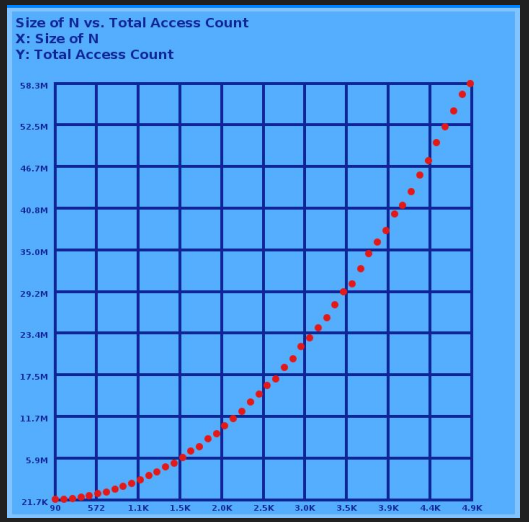
- **The plots below show the total access count as the size of the input (N) increases up to 5000 total elements.**

Best Case: O(N)



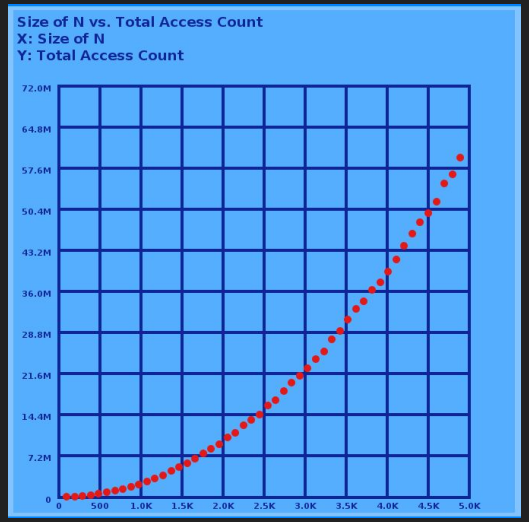
Happens when the input array is already sorted in ascending order.

Average Case: O(N^2)



Happens when the input array is made up of random values.

Worst Case: O(N^2)



Happens when the input array is sorted in descending (reverse) order.

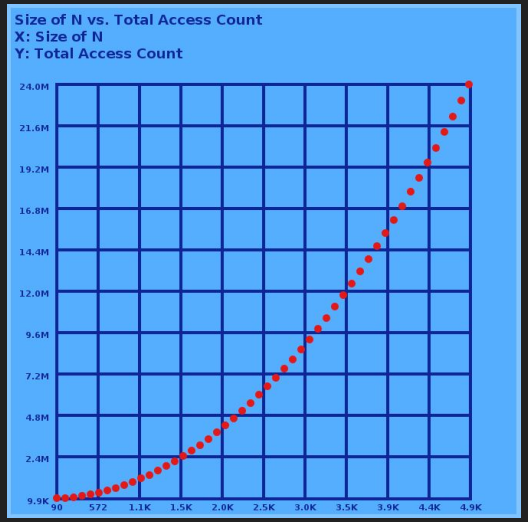
Selection Sort: Overview

- Selection Sort is a simple comparison-based sorting algorithm that works by repeatedly finding the minimum element (considering ascending order) from the unsorted part of the array and moving it to the beginning.
- During each pass through the array, it selects the smallest unsorted element and swaps it with the element in the next starting position.
- This process is repeated until the entire array is sorted, making it an intuitive but less efficient choice for large datasets due to its **$O(N^2)$** time complexity.
- Since Selection Sort is an in-place sorting algorithm, it has a space complexity of **O(1)**.

Selection Sort: Runtimes on Access Count Plots

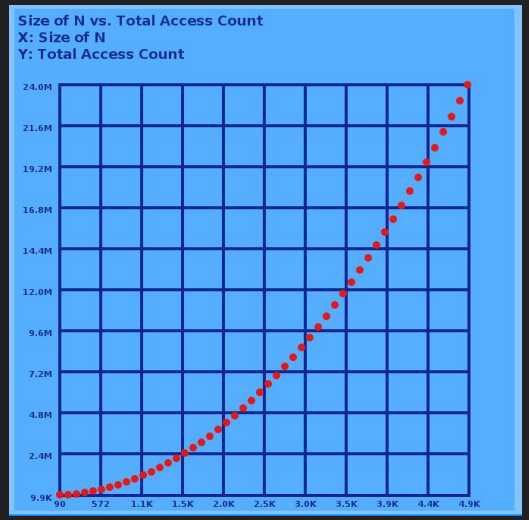
- **Since Selection Sort is not sensitive to the values of the input array when sorting, its best, average, and worst case runtimes are all the same: $O(N^2)$.**
- **To prove this, we still made multiple plots below with various types of data (ascending values, descending values, randomized values) for 5000 elements.**

Ascending Values



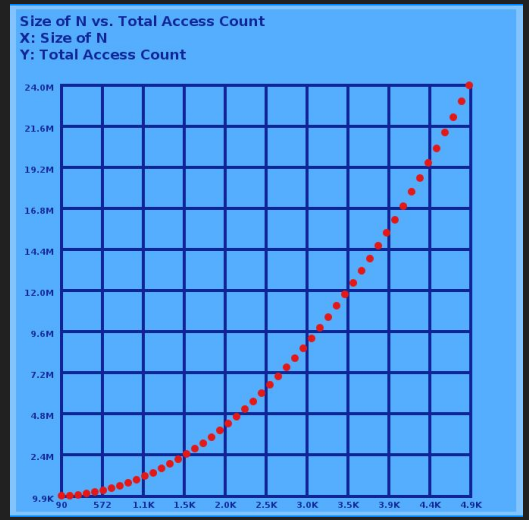
This is the "best case" for most sorting algorithms, but the access counts do not differ.

Random Values



This is the "average case" for most sorting algorithms, but the access counts do not differ.

Descending Values



This is the "worst case" for most sorting algorithms, but the access counts do not differ.

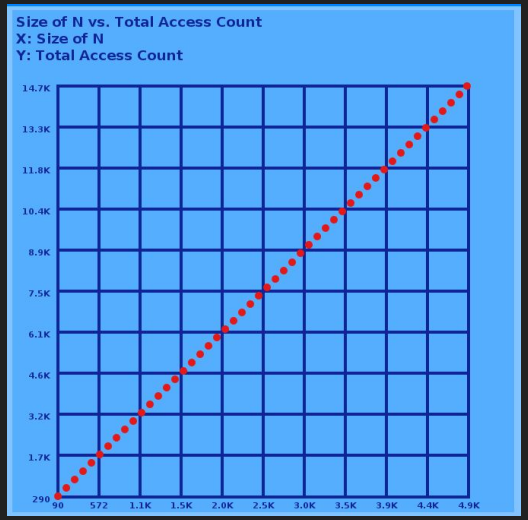
Insertion Sort: Overview

- This algorithm uses nested loops to iterate over each unsorted element in an array.
- The inner loop compares the current element with the already sorted elements, shifting them to make room for the current element. This process is repeated until all elements are sorted.
- The outer loop ensures that this inner loop occurs for every unsorted element.
- Since Insertion Sort is an in-place sorting algorithm, its space complexity is O(1).

Insertion Sort: Runtimes on Access Count Plots

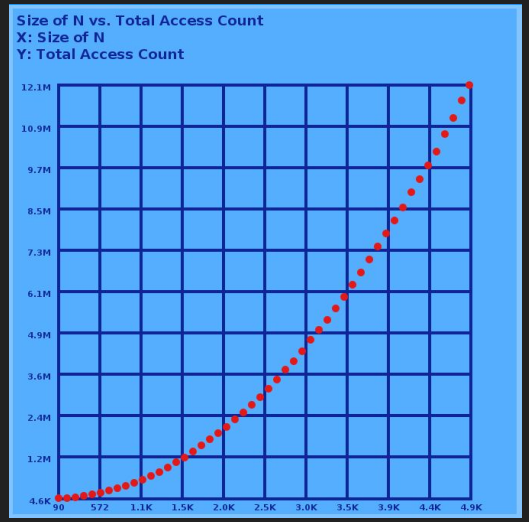
- **The plots below show the total access count as the size of the input (N) increases up to 5000 total elements.**

Best Case: O(N)



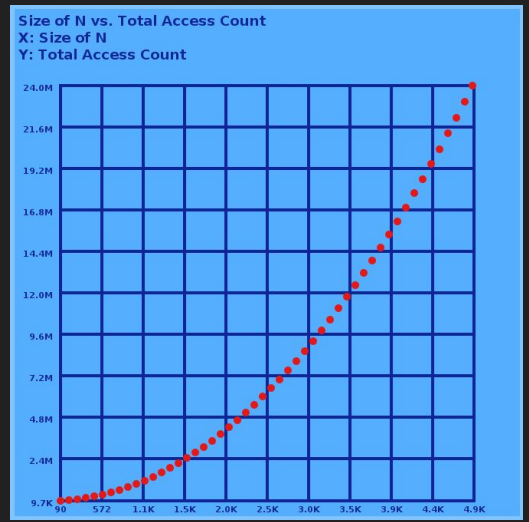
Happens when the input array is already sorted in ascending order.

Average Case: O(N^2)



Happens when the input array is made up of random values.

Worst Case: O(N^2)



Happens when the input array is sorted in descending (reverse) order.

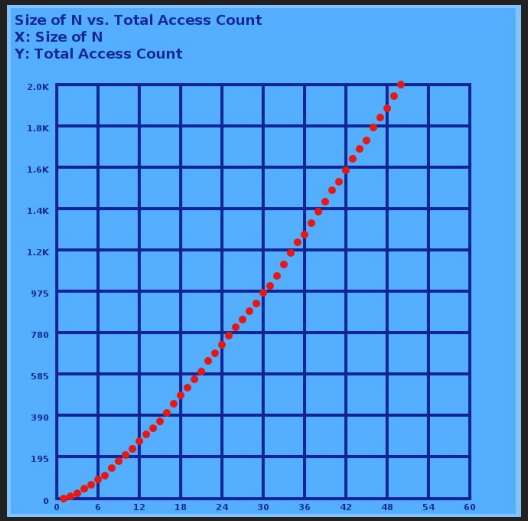
Heap Sort: Overview

- Heap Sort works by rearranging the array into a max-heap arrangement and then repeatedly inserting the largest heap elements into the final, sorted portion of the array.
- While an iterative version of Heap Sort would have **O(1)** for its space complexity, since we implemented our sink operation recursively, which has a runtime of **O(log(N))** so it puts **log(N)** calls on the call stack, the overall space complexity is **O(log(N)).**
- The runtime for heapsort is **O(NlogN)**, where the **log(N)** comes from the heapify/sink method and the **N** from swapping the max **N-1** times.

Heap Sort: Runtimes on Access Count Plots

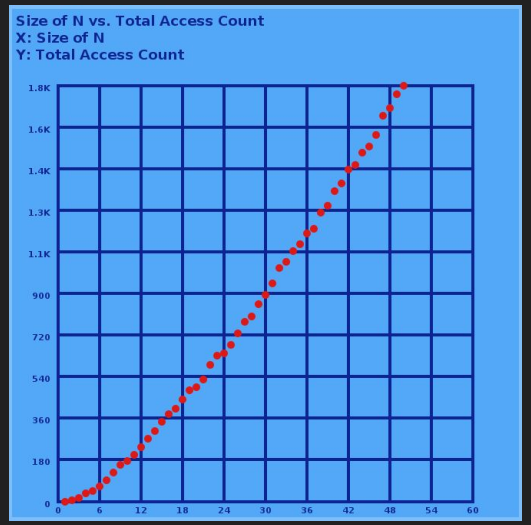
- Heap Sort will always have a runtime of O(N log(N)). Regardless of the element values, the largest value in the heap will need to be swapped into the sorted result N-1 times, where the runtime of sinking the topmost heap element is O(log(N)), thus the overall runtime is O(N log(N)).
- To prove this, we still made multiple plots below with various types of data (ascending values, descending values, randomized values).
- We used 50 elements instead of 5000 so the logarithmic curve is more noticeable.
- NOTE: the access count will be slightly different because the values may affect the arrangement of the heap elements and how far they need to be sunk, but the curve is still O(N log(N)).

Ascending Values



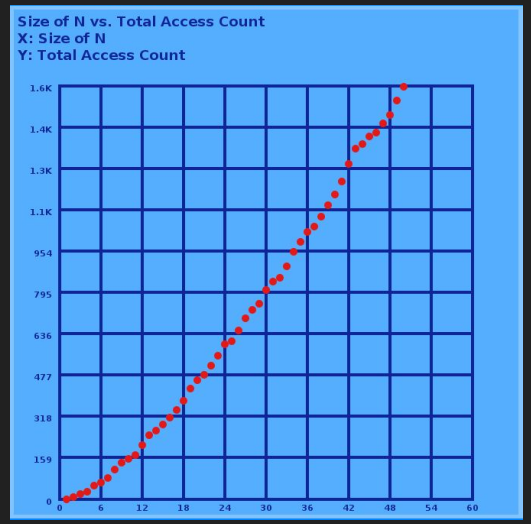
This is the "best case" for most sorting algorithms, but the shape of the curve does not differ.

Random Values



This is the "average case" for most sorting algorithms, but the shape of the curve does not differ.

Descending Values



This is the "worst case" for most sorting algorithms, but the shape of the curve does not differ.

Quick Sort: Overview

- Quick Sort works by choosing an element as a "pivot" to partition the array into two groups in **O(N)** time: elements less than (and swapped to the left of) the pivot and elements greater than (and swapped to the right of) the pivot.
- It then makes two recursive calls, one for each group, and continues the process until the whole input array is sorted.
- The goal is to choose the **median** as the pivot each time to always create equally-sized groups so the input is halved to ultimately make **log(N)** calls and run in **O(N log(N))** time instead of **N** calls and running in **O(N^2)** time.

Quick Sort: Runtime In-Depth

- The fundamental worst-case and best-case situations of Quick Sort is the same for all implementations: if the pivot is chosen as the minimum or maximum each time, then the runtime is **O(N^2)**, but if it is the median each time, the runtime is **O(N log(N))**.
- However, it is how an implementation choses its **pivot** that affects what **input array** would trigger these cases: since our version choses the middlemost element instead of the first element like the in-class version, our worst and best **input** will be different.

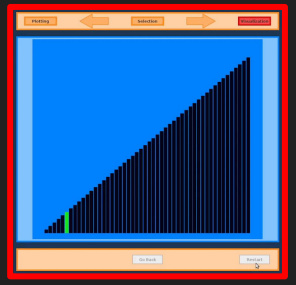
```
if (left < right) {
    int pivotIndex = (left + right) / 2;
    int pivot = array.get(pivotIndex);
    int i = left, j = right;

    while (i <= j) {
        while (array.get(i) < pivot)
            i++;
        while (array.get(j) > pivot)
            j--;
        if (i <= j) {
            swap(array, i, j);
            i++;
            j--;
        }
    }
    if (left < j)
        helperQuickSort(group, array, left, j);
    if (i < right)
        helperQuickSort(group, array, i, right);
}
```

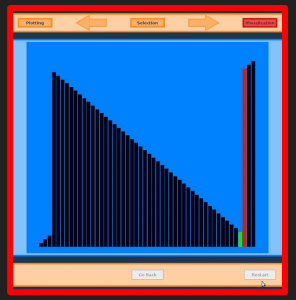
BEST CASE: ASCENDING

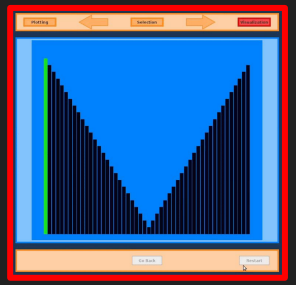
- Since everything to the left of the median (pivot) is less than the pivot and everything to the right is greater, the indices **I** and **J** will meet each other before anything is swapped.
- Although the pivot is always the median in descending order, it still needs to swap the larger and smaller elements around.

WORST CASE:
MIDDLEMOST ELEMENT IS MAX/MIN

- We generated an input array with a "V" shape where the middlemost element for the first series of recursions is the minimum for the worst-case.





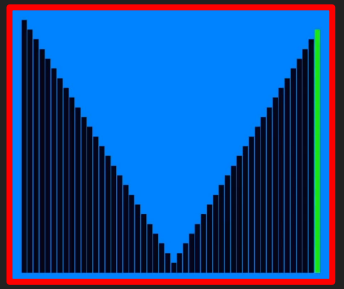


Quick Sort: Runtimes on Access Count Plots

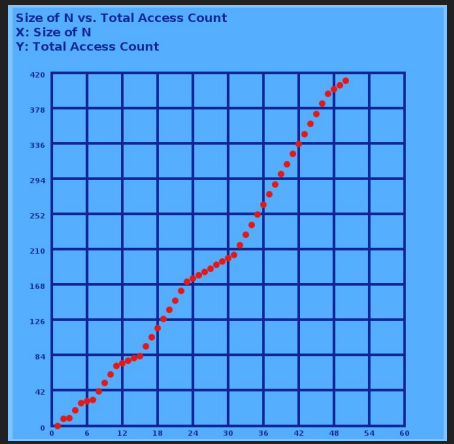
More on the **worst-case** runtime:

- We generated an input array of 50 elements with a "V" shape so that the smallest values are chosen as the pivot for each of the recursive calls (or as many as possible).
- This generated an access count of **~1,000**, which is higher than the average of **~750** for 50 elements of random values.

Again, we chose 50 elements instead of 5000 or above so the logarithmic curve was more obvious (otherwise it would look linear).

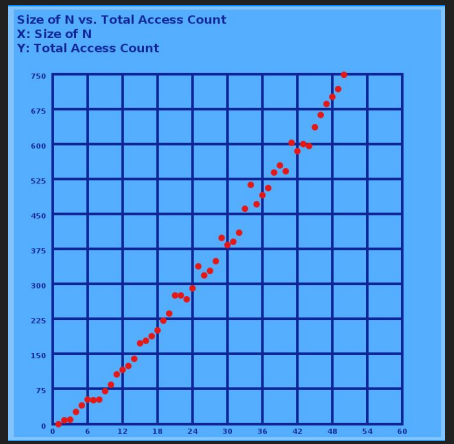


Best Case: O(N log(N))



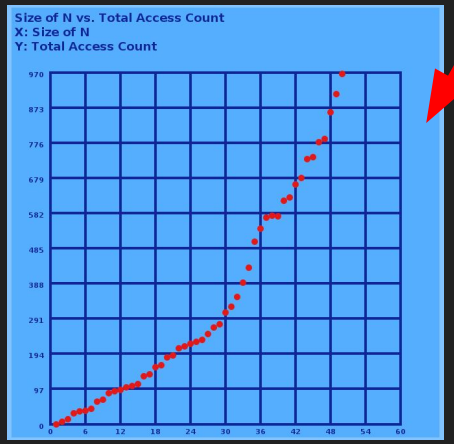
Happens when the input array is already sorted in ascending order.

Average Case: O(N log(N))



Happens when the input array is made up of random values.

Worst Case: O(N^2)



Happens when the input array has the middlemost value as the min each time.

Merge Sort: Overview

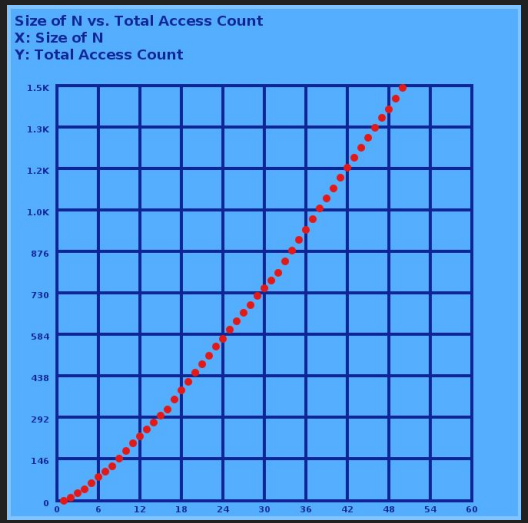
This algorithm recursively splits the array until each subarray is just 1 element. It then merges the subarrays back together in sorted order using a temporary array. Finally, it copies the temporary array back to the original array.

The space usage for Merge Sort is O(N) for the temporary array and its runtime for best and worst case is O(NlogN), where the N comes from the merge method and the logN is the amount of times you merge the subarrays.

Merge Sort: Runtimes on Access Count Plots

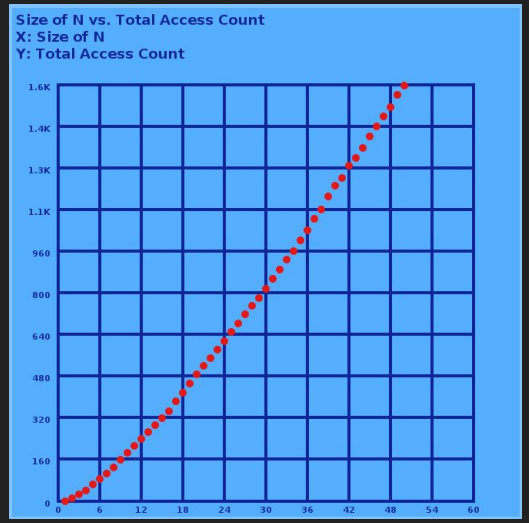
- Merge Sort will always have a runtime of O(N log(N)) because halving the array does not depend on the element values and the merging process will eventually iterate through each element of left-hand and right-hand sub-arrays once regardless of the element values.
- To prove this, we still made multiple plots below with various types of data (ascending values, descending values, randomized values).
- We used 50 elements instead of 5000 so the logarithmic curve is more noticeable.
- NOTE: the access count will be slightly different since our implementation performs more work to merge random values, but the shape is O(N log(N)) nonetheless.

Ascending Values



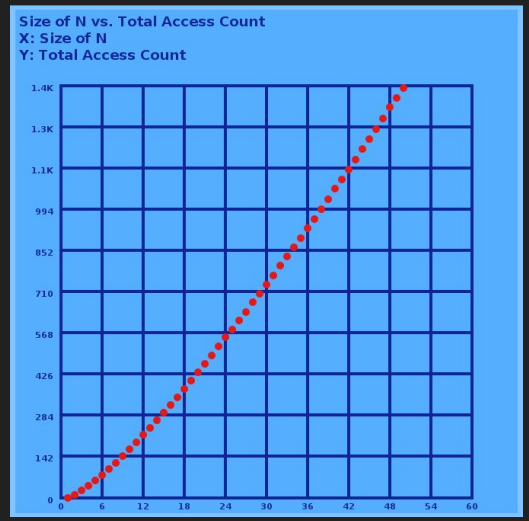
This is the "best case" for most sorting algorithms, but the shape of the curve does not differ.

Random Values



This is the "average case" for most sorting algorithms, but the shape of the curve does not differ.

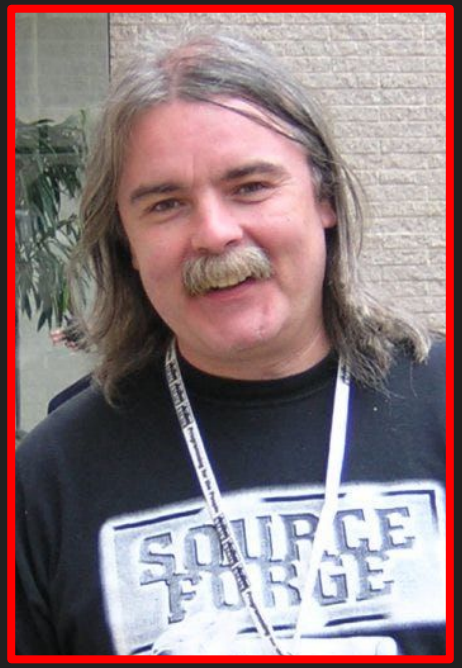
Descending Values



This is the "worst case" for most sorting algorithms, but the shape of the curve does not differ.

Merge-Based Hybrid Algorithms

- One very famous hybrid algorithm is **Timsort**, which was made by Tim Peters in 2002 as a combination of **Merge Sort** and **Insertion Sort** and has been in Python since version 2.3 (Auger, Nicolas, et al.).
- Inspired by how this algorithm utilizes **Insertion Sort's** strength on small sub-arrays, we decided to use hybrid algorithms that combine **Merge Sort** with other standalone algorithms.
- This is to see if we can improve upon the speed of the slower algorithms (**Bubble, Selection**) and see if the faster algorithms can improve on **Merge Sort's** existing runtime (**Heap, Quick**).

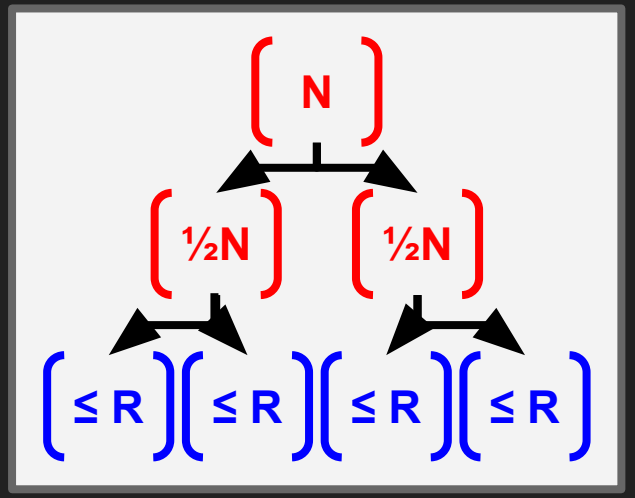


Tim Peters

(from *Medium*)

Hybrid Algorithms: Math for Runtime

- Each of our merge-based algorithms is similar to regular Merge Sort by **(1) recursively dividing the array into halves** and **(2) eventually merging sorted sub-arrays**.
    - However, when the input is split into sub-arrays of a size less than our **threshold value (R)**, the hybrid algorithm calls the standalone algorithm to sort those R elements.



- Recall that the runtime of Merge Sort is **O(N log(N))** because it makes **log(N)** recursive calls that need to merge two sub-arrays in **O(N)** time.
- Here is the modified runtime taking the threshold **R** into account and the runtime **T(R)** of the standalone algorithm on **R**:

$$N(\log N - \log R) + \frac{N}{R}(T(R))$$

- There are now **log(N) - log(R)** calls that divide and merge the input (since there are **log(R)** less recursions needed) and **N/R** sub-arrays that are sorted with the standalone algorithm.

Hybrid Algorithms: Math for Runtime

- To apply the new runtime formula to our hybrid algorithms, consider that the best, average, and worst cases of the standalone algorithms involves one of these three runtimes: **O(N)**, **O(N^2)**, and **O(N log(N))**.
- This means that **T(R)** may be either **O(R)**, **O(R^2)**, and **O(R log(R))**, which we can substitute into our formulas so we can further simplify them to represent specific runtimes.
- However, since our hybrid algorithms will all set **R=10** for simplicity, it can be treated as a constant for the Big-Oh notation. Because of this, the runtime of our hybrid algorithms will all be **O(N log(N))**.

FOR RUNTIME OF: $O(R^2)$
For any R:

$$N(\log N - \log R) + \frac{N}{R}(R^2)$$
$$= N\log(N) - N\log(R) + NR$$
... in Big-Oh: $O(N\log(N) + NR)$

When R is set to a constant 10:
$N\log(N) - N\log(10) + N(10)$
... in Big-Oh:
$O(N\log(N))$

FOR RUNTIME OF: $O(R)$
For any R:

$$N(\log N - \log R) + \frac{N}{R}(R)$$
$$= N\log(N) - N\log(R) + N$$
... in Big-Oh: $O(N\log(N) - N\log(R))$

When R is set to a constant 10:
$N\log(N) - N\log(10) + N$
... in Big-Oh:
$O(N\log(N))$

FOR RUNTIME OF: $O(R\log R)$
For any R:

$$N(\log N - \log R) + \frac{N}{R}(R\log R)$$
$$= N\log(N) - N\log(R) + N\log(R)$$
... in Big-Oh: $O(N\log(N))$

When R is set to a constant 10:
$N\log(N)$ (there is no R)
... in Big-Oh: $O(N\log(N))$

Hybrid Algorithms: Math for Space Complexity

- Although **Merge Sort** already requires a temporary array with a space complexity of **O(N)**, there are two recursive algorithms that occupy space on the call stack: **Quick Sort** and **Heap Sort**.
- Our **Quick Sort** makes **log(R)** recursive calls on average, so its space complexity is **O(log(N))**.
- An iterative **Heap Sort** would have a space complexity of **O(1)**, but since our implementation has a recursive **`heapify()`** or **`sink()`** operation that makes **log(N)** calls on average, our **Heap Sort** also has a space complexity of **O(log(N))**.

Here is the general formula where **T(R)** is the space complexity of the **standalone** algorithm:

$$N + (\log N - \log R) + T(R)$$

In terms of **Merge Sort**, there are **N** elements in its temporary array and **log(N) - log(R)** recursive calls made. Although there are **N/R** sub-arrays and thus **N/R** times the **standalone** algorithm is called, **Merge Sort** will only run one instance of the **standalone** at a time. The copies of **T(R)** do not add upon each other over time because **T(R)** represents the instance's stack frames that are removed once it finishes sorting.

Hybrid Algorithms: Math for Space Complexity

- Like with the runtime formula, since our hybrid algorithms have **R=10**, **R** is effectively a constant and is dropped in the Big-Oh notation.

FOR SPACE COMPLEXITY OF: $O(\log(R))$

For any R:

$N+(\log N-\log R)+\log(R)$

$=N+\log(N)$

... in Big-Oh: $O(N)$

When R is set to a constant 10:

$=N+\log(N)$ (there is no R)

... in Big-Oh: $O(N)$

FOR SPACE COMPLEXITY OF: $O(1)$

For any R:

$N+(\log N-\log R)$

... in Big-Oh: $O(N-\log(R))$

When R is set to a constant 10:

$=N+(\log N-\log(10))$

... in Big-Oh: $O(N)$

Hybrid Algorithms: Summary of Complexity

= Standalone Complexity = Hybrid Complexity (Including R) = Hybrid Complexity (R is Constant)

	Best-Case Runtime	Average Runtime	Worst-Case Runtime	Space Complexity
Bubble	$O(N)$ $O(N \log(N) - N \log(R))$ $O(N \log(N))$	$O(N^2)$ $O(N \log(N) + NR)$ $O(N \log(N))$		$O(1)$ $O(N - \log(R))$ $O(N)$
Insertion	$O(N)$ $O(N \log(N) - N \log(R))$ $O(N \log(N))$	$O(N^2)$ $O(N \log(N) + NR)$ $O(N \log(N))$		$O(1)$ $O(N - \log(R))$ $O(N)$
Selection	$O(N^2)$ $O(N \log(N) + NR)$ $O(N \log(N))$			$O(1)$ $O(N - \log(R))$ $O(N)$
Heap	$O(N \log(N))$ $O(N \log(N))$ $O(N \log(N))$			$O(\log(R))$ $O(N)$ $O(N)$
Quick	$O(N \log(N))$ $O(N \log(N))$ $O(N \log(N))$			$O(\log(R))$ $O(N)$ $O(N)$

Bubble-Merge: Overview

- This algorithm combines the Bubble Sort and Merge Sort algorithms. It recursively partitions the array until the size of the subarray size becomes 10 or smaller.
- For subarrays of size 10 or less, it performs a quicksort. For larger subarrays, the algorithm divides the array, applies the `bubbleMerge()` recursively on each half, and then merges the sorted halves.
- In order to determine the space complexity, we consider the space complexity for both Bubble Sort and Merge Sort. See the next slide for a detailed explanation.

Bubble-Merge: Space Complexity

The space is determined by O(**N** + **((log(N) - log(R))**) where:

- The outer **N** is the temporary array for merge
- **R** is the smallest subarray merge will split
- **((log(N) - log(R))** is the number of calls made before the hybrid algorithm performs Bubble Sort, but the **log(N)** is dominated by **N**

The space complexity is **O(N - log(R))**. Since the **R** is set to a constant of 10, is becomes **O(N - log(10))** which is really **O(N)**.

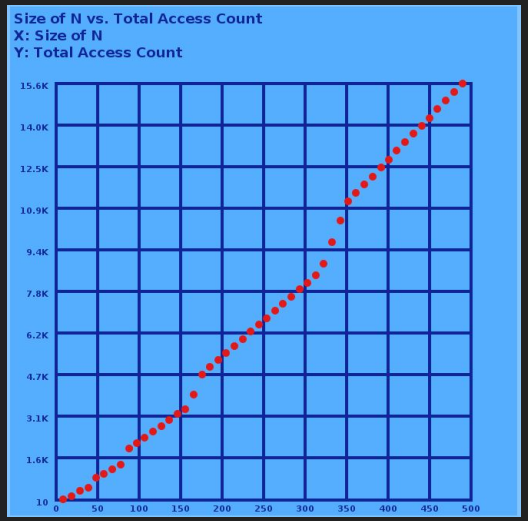
Bubble-Merge: Runtime

1. Bubble Sort has a separate best-case and worst-case runtime, so Bubble-Merge will also have a best-case and worst-case runtime.
2. For both cases, the merge portion will always add to the runtime with **N(log(N) - log(R))** due to the **log(N) - log(R)** calls and the O(**N**) merge operation.
3. For the worst-case, the Bubble part of Bubble-Merge will add to the runtime with **(N/R)(R^2)** since it is ran **(N/R)** times, each with a runtime of O(**R^2**). Altogether, the worst-case runtime of Bubble-Merge is **O(Nlog(N) + NR)**, which simplifies to **O(Nlog(N))** due to the constant **R=10** for our implementations.
4. For the best-case, the Insertion part of Bubble-Merge will add to the runtime with **(N/R)(R)** since it is ran **(N/R)** times, each with a runtime of O(**R**). Altogether, the best-case runtime of Bubble-Merge is **O(Nlog(N) - Nlog(R))**, which simplifies to **O(Nlog(N))** due to the constant **R=10** for our implementations.

Bubble-Merge: Runtimes on Access Count Plots

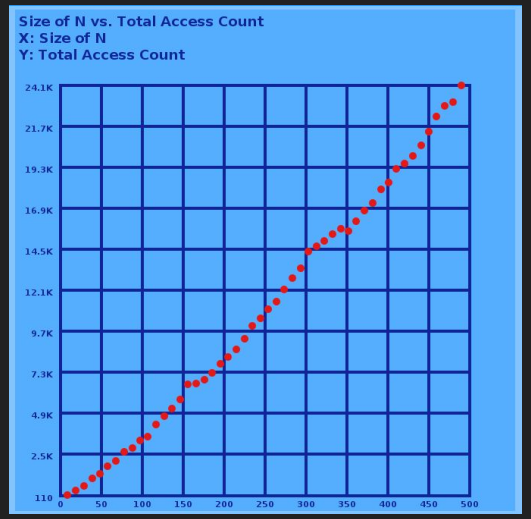
- As proven by our mathematical analysis for the hybrid algorithms, Bubble-Merge will always have a runtime of O(N log(N)).
- To prove this, we still made multiple plots below with various types of data (ascending values, descending values, randomized values).
- We used 500 elements this time to produce a more stable curve than 50 elements that is still noticeably logarithmic.
- NOTE: the access count may vary slightly due to the merging process and the behavior of Bubble Sort.

Ascending Values



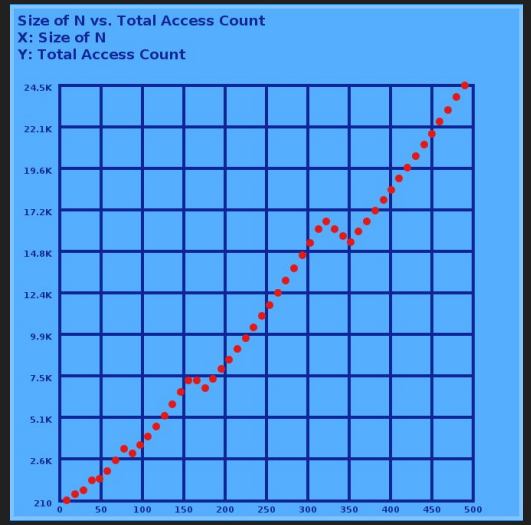
This is the "best case" for most sorting algorithms, but the shape of the curve does not differ.

Random Values



This is the "average case" for most sorting algorithms, but the shape of the curve does not differ.

Descending Values



This is the "worst case" for most sorting algorithms, but the shape of the curve does not differ.

Merge-Selection: Overview

- Merge-Selection Sort divides the input array into halves like Merge Sort and sorts sub-arrays of a size below a threshold R with Seleciton Sort. For our implementation, we fixed the threshold to 10.
- It was implemented with a `mergeSelection()` method that recursively calls itself and calls the same `selectionSort()` method as Selection Sort on the sub-arrays of size R and merges them back together with the same `merge()` method as Merge Sort.
- The runtime for this algorithm is **O(Nlog(N))** and it is explained more in detail in the next slide.

Merge-Selection: Runtime

1. Selection Sort always has runtime of $O(N^2)$, so Merge-Selection will only have one runtime as well.
2. The merge portion will always add to the runtime with **N(log(N) - log(R))** due to the **log(N) - log(R)** calls and the O(**N**) merge operation.
3. The Selection part of Merge-Selection will add to the runtime with **$(N/R)(R^2)$** since it is ran **(N/R)** times, each with a runtime of $O(R^2)$. Altogether, the runtime of Merge-Selection is **O(Nlog(N) + NR)**, but since our **R** is fixed to 10 for our implementations, the runtime is ultimately **O(Nlog(N))**.

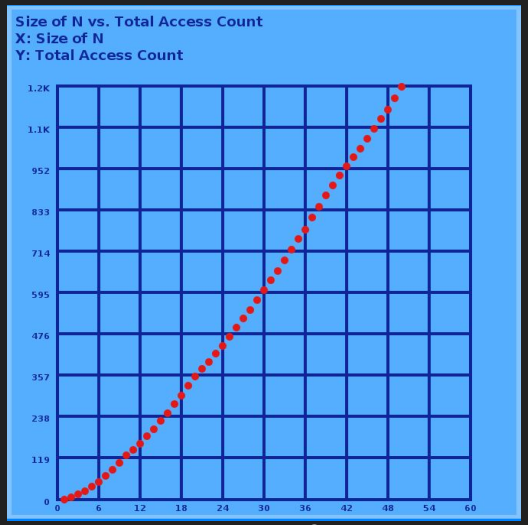
Merge-Selection: Space Complexity

- Since our Selection Sort does not rely on any auxiliary data structure nor does it make any recursive calls to itself, its space complexity is **O(1)**.
- Because of this, the space complexity of Merge-Selection comes from Merge Sort, which is **N+(log(N)-log(R))** due to the temporary array of size **N** and the **log(N)-log(R)** recursive calls, which simplifies to **O(N-log(R))**.
- However, since our **R** is fixed to 10 for our our implementations, the space complexity is ultimately **O(N)**.

Merge-Selection: Runtimes on Access Count Plots

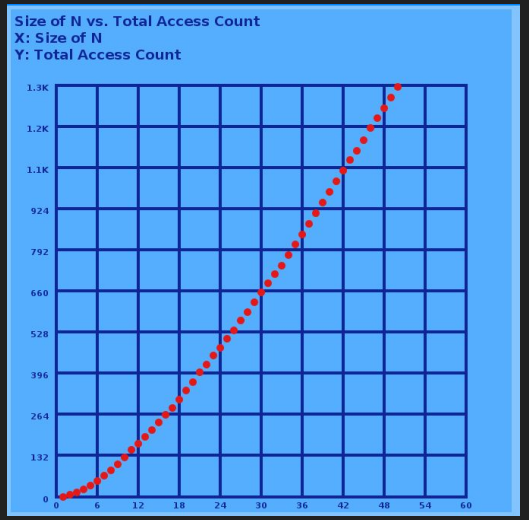
- As proven by our mathematical analysis for the hybrid algorithms, Merge-Selection will always have a runtime of O(N log(N)).
- To prove this, we still made multiple plots below with various types of data (ascending values, descending values, randomized values).
- Since Selection Sort is predictable in its behavior, we lowered the element count back to 50 to emphasize the logarithmic curve.
- NOTE: the access count may vary slightly due to the merging process, although we know Selection Sort does not normally depend on the element values.

Ascending Values



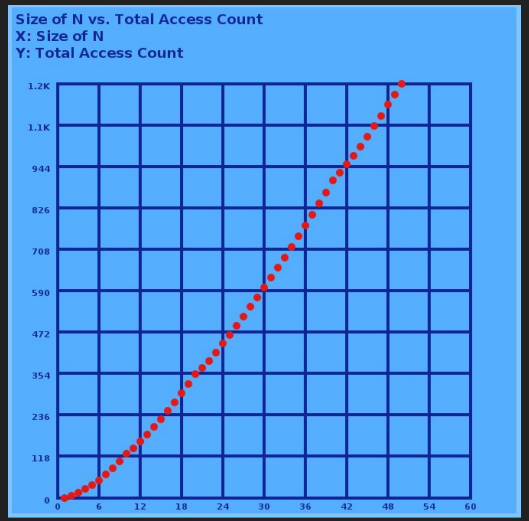
This is the "best case" for most sorting algorithms, but the shape of the curve does not differ.

Random Values



This is the "average case" for most sorting algorithms, but the shape of the curve does not differ.

Descending Values



This is the "worst case" for most sorting algorithms, but the shape of the curve does not differ.

Merge-Insertion: Overview

- Merge-Insertion Sort divides the input array into halves like Merge Sort and sorts sub-arrays of a size below a threshold R with Insertion Sort. For our implementation, we fixed the threshold to 10.
- It was implemented with a `mergeInsertion()` method that recursively calls itself and calls the same `insertionSort()` method as Insertion Sort on the sub-arrays of size R and merges them back together with the same `merge()` method as Merge Sort.
- The runtime for this algorithm is **O(Nlog(N))** and it is explained more in detail in the next slide.

Merge-Insertion: Runtime

1. Insertion Sort has a separate best-case and worst-case runtime, so Merge-Insertion will also have a best-case and worst-case runtime.
2. For both cases, the merge portion will always add to the runtime with **N(log(N) - log(R))** due to the **log(N) - log(R)** calls and the O(**N**) merge operation.
3. For the worst-case, the Insertion part of Merge-Insertion will add to the runtime with **(N/R)(R^2)** since it is ran **(N/R)** times, each with a runtime of O(**R^2**). Altogether, the worst-case runtime of Merge-Insertion is **O(Nlog(N) + NR)**, which simplifies to **O(Nlog(N))** due to the constant **R=10** for our implementations.
4. For the best-case, the Insertion part of Merge-Insertion will add to the runtime with **(N/R)(R)** since it is ran **(N/R)** times, each with a runtime of O(**R**). Altogether, the best-case runtime of Merge-Insertion is **O(Nlog(N) - Nlog(R))**, which simplifies to **O(Nlog(N))** due to the constant **R = 10** for our implementations.

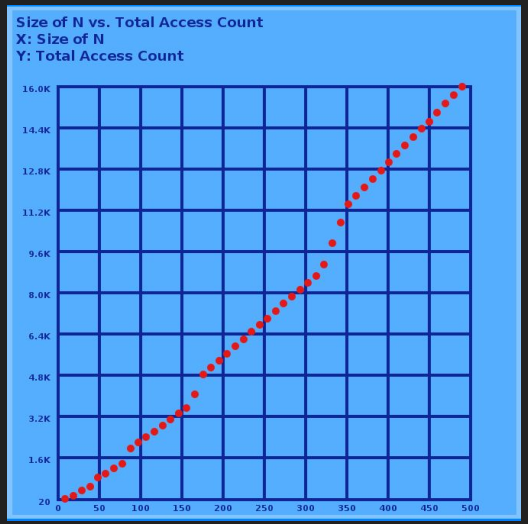
Merge-Insertion: Space Complexity

- Since our Insertion Sort does not rely on any auxiliary data structure nor does it make any recursive calls to itself, its space complexity is **O(1)**.
- Because of this, the space complexity of Merge-Insertion comes from Merge Sort, which is **N+(log(N)-log(R))** due to the temporary array of size **N** and the **log(N)-log(R)** recursive calls, which simplifies to **O(N-log(R)**.
- However, since our **R** is fixed to 10 for our our implementations, the space complexity is ultimately **O(N)**.

Merge-Insertion: Runtimes on Access Count Plots

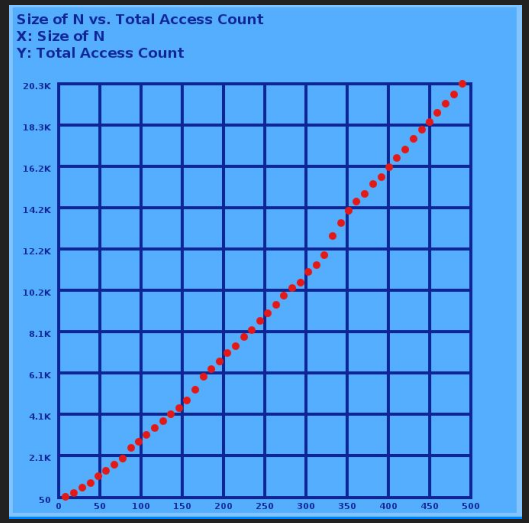
- As proven by our mathematical analysis for the hybrid algorithms, Merge-Insertion will always have a runtime of O(N log(N)).
- To prove this, we still made multiple plots below with various types of data (ascending values, descending values, randomized values).
- We used 500 elements this time to produce a more stable curve than 50 elements that is still noticeably logarithmic.
- NOTE: the access count may vary slightly due to the merging process and the behavior of Insertion Sort.

Ascending Values



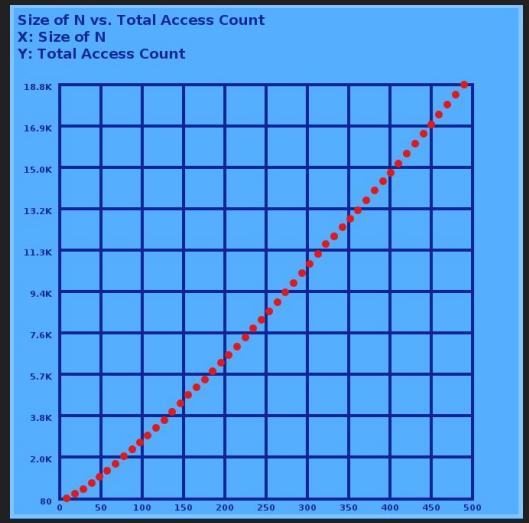
This is the "best case" for most sorting algorithms, but the shape of the curve does not differ.

Random Values



This is the "average case" for most sorting algorithms, but the shape of the curve does not differ.

Descending Values



This is the "worst case" for most sorting algorithms, but the shape of the curve does not differ.

Heap-Merge: Overview

- Heap-Merge Sort is an efficient sorting algorithm that recursively splits the array until it reaches a threshold, **R**, calls Heap Sort on each sublist, then merges them back together with the regular merge algorithm.
- It was implemented with a `heapMerge()` method that recursively calls itself and calls the same `heapSort()` method as Heap Sort on the sub-arrays of size R and merges them back together with the same `merge()` method as Merge Sort.
- The runtime for this algorithm is **O(Nlog(N))** and it is explained more in detail in the next slide.

Heap-Merge: Runtime

1. We know that we have a threshold, R, and this is the length of our subarrays that we call Heap Sort on. The runtime of regular Heap Sort is **O(N log(N))**, so in terms of R, the runtime for sorting a sub-array is **Rlog(R)**, and we would have **N/R** of these subarrays. Therefore, our Heap Sort runtime for this hybrid algorithm is **(N/R)(Rlog(R))**.
2. We know that merge has a runtime of **O(N)** and the amount of times we would merge would be **log(N) - log(R)** since we want to stop at our threshold. So our merge runtime of this hybrid algorithm is **N(log(N) - log(R))**.
3. With the runtime of Merge Sort and Heap Sort combined, the overall runtime is **O(N(log(N) - log(R)) + (N/R)(Rlog(R)))** which simplifies to **O(Nlog(N))** due to the constant **R = 10** for our implementations.

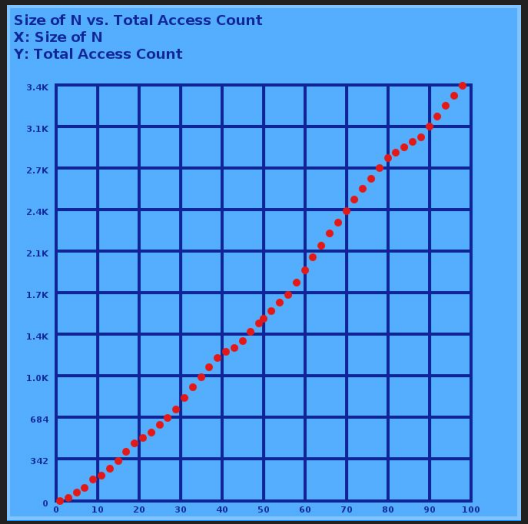
Heap-Merge: Space Complexity

- If we were to implement Heap Sort iteratively, the space complexity would be **O(1)**, but since our sink operation is recursive and not iterative and it makes **log(N)** calls to rearrange elements to the bottom of the heap, the overall space complexity is **O(log(N))** — this is written as **log(R)** in the formula.
- The merge portion of the space complexity is **N + (log(N) - log(R))** because of the temporary array of size **N** and the **log(N) - log(R)** recursive calls.
- The positive and negative **log(R)** cancel out, leaving only **N + log(N)**. Since the **log(N)** is dominated by **N**, the final space complexity for the whole hybrid algorithm is **O(N)**.

Heap-Merge: Runtimes on Access Count Plots

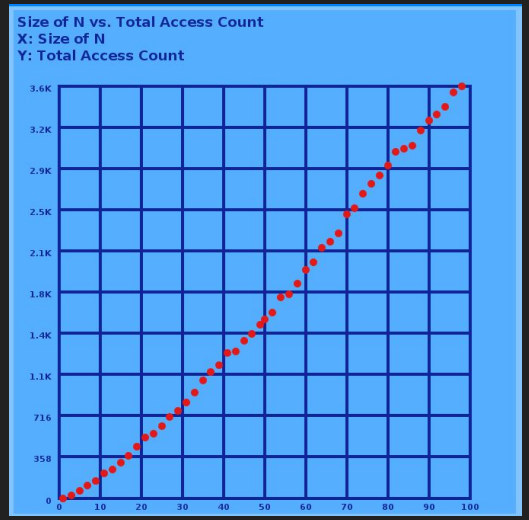
- As proven by our mathematical analysis for the hybrid algorithms, Heap-Merge will always have a runtime of O(N log(N)).
- To prove this, we still made multiple plots below with various types of data (ascending values, descending values, randomized values).
- We used 100 elements this time to produce a more stable curve than 50 elements that is still noticeably logarithmic.
- NOTE: the access count may vary slightly due to the merging process and the behavior of Heap Sort.

Ascending Values



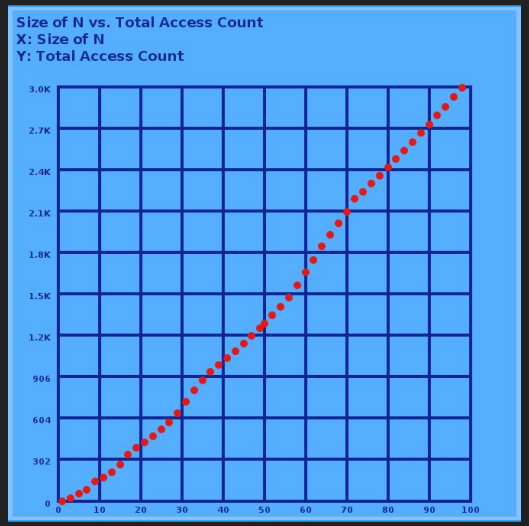
This is the "best case" for most sorting algorithms, but the shape of the curve does not differ.

Random Values



This is the "average case" for most sorting algorithms, but the shape of the curve does not differ.

Descending Values



This is the "worst case" for most sorting algorithms, but the shape of the curve does not differ.

Quick-Merge: Overview

- This algorithm combines the Quick Sort and Merge Sort algorithms. It recursively partitions the array until the size of the subarray (right - left + 1) becomes 10 or smaller.
- For subarrays of size 10 or less, it performs a Quick Sort. For larger subarrays, it divides the array into two halves, recursively calls `quickMerge()` on each half, and then merges the sorted halves.
- In order to determine the space complexity, we consider the space complexity for both Quicksort and Merge Sort. See the next slide for a detailed explanation.

Quick-Merge: Space Complexity

The space is determined by O(**N** + **((log(N) - log(R))** + **log(R)**) where:

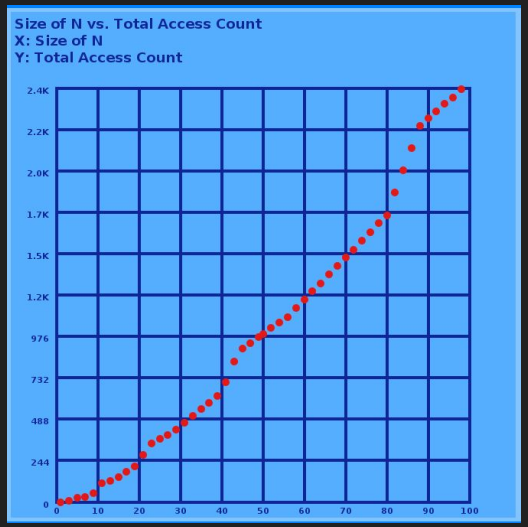
- The outer **N** is the temporary array for merge
- **R** is the smallest subarray merge will split
- **((log(N) - log(R))** is the number of calls made before the hybrid algorithm performs Quick Sort, but it is dominated by **N** and cancels out with **log(R)**
- **log(R)** is the number of recursive calls Quick Sort makes

This simplifies to **O(N)**.

Quick-Merge: Runtimes on Access Count Plots

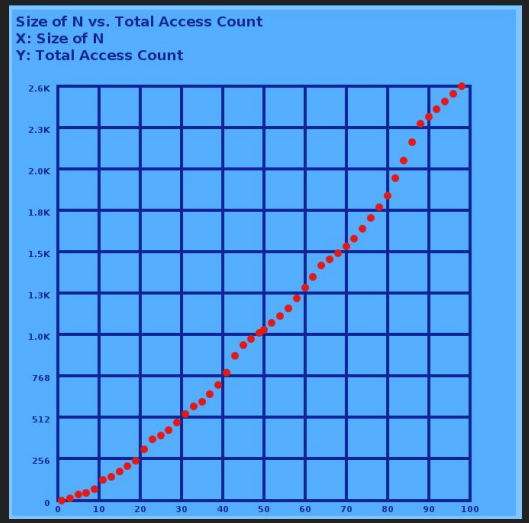
- As proven by our mathematical analysis for the hybrid algorithms, Quick-Merge will always have a runtime of O(N log(N)).
- To prove this, we still made multiple plots below with various types of data (ascending values, descending values, randomized values).
- We used 100 elements this time to produce a more stable curve than 50 elements that is still noticeably logarithmic.
- NOTE: the access count may vary slightly due to the merging process and the behavior of Quick Sort.

Ascending Values



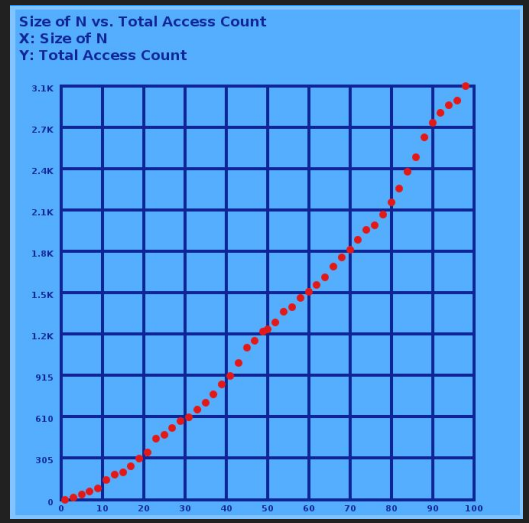
This is the "best case" for most sorting algorithms, but the shape of the curve does not differ.

Random Values



This is the "average case" for most sorting algorithms, but the shape of the curve does not differ.

Descending Values



This is the "worst case" for most sorting algorithms, but the shape of the curve does not differ.

Conclusion

Overall, from our theoretical and experimental analysis with the math and plots, hybrids created with Merge Sort are a simple yet effective way of balancing runtime and space complexity.

Strengths of Merge-Based Hybrid Algorithms

- The benefits of Merge Sort are most pronounced with the simpler, slower sorting algorithms like Bubble, Insertion, and Selection Sort that have poor runtime yet ideal space complexity.
- As seen in our mathematical analysis on slides 18-22, all of these algorithms were brought to an **O(Nlog(N))** runtime, which could vary depending on how R is defined. This even applies to Selection Sort, an algorithm that is consistently slow regardless of the input.
- Although this also means that these algorithms that are otherwise O(1) in space complexity are now O(N), this can be improved in real-world runtimes with large values of **R** that can reduce the **N** size complexity by **log(R)** in the Big-Oh formula of **O(N - log(R))**.

Weaknesses of Merge-Based Hybrid Algorithms

- On the other hand, combining Merge Sort with algorithms that are already recursive like Quick Sort and Heap Sort actually may introduce additional overhead and diminishing results.
- In the mathematical runtime, the **Nlog(R)** savings from the base Merge Sort runtime cancels out with the **Nlog(R)** cost of running the standalone algorithm, which cancels out to **Nlog(N)** anyway.
- Making matters worse, the **O(log(N))** space complexity of these standalone algorithms are now at least **O(N)**.

Works Cited

Auger, Nicolas, et al. *On the Worst-Case Complexity of TimSort*, 22 May 2018, doi.org/10.48550/arXiv.1805.08612.

Scheiwe, Richard. “The Case for Timsort.” *Medium*, 24 Aug. 2018, medium.com/@rscheiwe/the-case-for-timsort-349d5ce1e414.